MICROCHIP

AN852

Implementing FIR and IIR Digital Filters Using PIC18 Microcontrollers

Author: *B. K. Anantha Ramu*
Microchip Technology Designs (India) Pvt. Ltd.

INTRODUCTION

The Microchip PICmicro® PIC18 family of microcontrollers are popularly known for their logic and controlling functions. In addition, these microcontrollers have built-in hardware multipliers and multiple file pointers. These features, along with the built-in analog-to-digital converter (ADC), make PIC18 microcontrollers a competent choice for applications where logic and controlling functions are combined with signal processing applications.

This application note demonstrates how the PIC18 family of microcontrollers can be used to implement digital FIR and IIR filters.

Note: This application note assumes the reader understands the basics of digital filters and their types. Refer to Appendix A should you require additional information.

The process of building a digital filter involves the following two distinct phases:

- Design phase
- Realization phase

Design Phase

The design phase involves specifying filter characteristics (e.g., frequency response, phase response, etc.) and deriving the input output transfer function or filter coefficients from the specifications. Many software tools are available to generate filter coefficients from the specified filter characteristics.

Realization Phase

The realization phase involves the selection of a structure to implement the transfer function. The structure may be a circuit if the filter is built by hardware, or may be a software program if implemented on microcontrollers.

FIR FILTER IMPLEMENTATION

Equation 1 shows the computation performed by an FIR filter.

EQUATION 1: Y[N] COMPUTATION

$$y[n] = x[n]*a_0+x[n-1]*a_1+x[n-2]*a_2+....+ x[n-N+1]*a_{N-1}$$

Where N is the number of taps and $a_0, a_1, \ldots a_{N-1}$ are N filter coefficients. The N filter coefficients can be positive or negative depending on the characteristics of the filter. The computation performed by an FIR filter is implemented in a PIC18 microcontroller in two stages.

First, the output value $y^1[n]$ is computed using the formula shown in Equation 2.

EQUATION 2: $Y^1[N]$ COMPUTATION

$$y^1[n] = x[n]*(a_0 + 128) + x[n-1]*(a_1 + 128) + x[n-2]*(a_2 + 128)+...+x[n-N+1]*(a_{N-1} + 128)$$

Second, X[n], as shown in Equation 3, is subtracted from $y^1[n]$ to obtain y[n].

EQUATION 3: X[n] COMPUTATION

$$X[n] = x[n]*128 + x[n-1]*128 +...+ x[n-N+1]*128$$

X[n] represents the sum of all input samples from the latest to the previous N–1 samples, multiplied by 128.

In calculating $y^1[n]$, we have added 128 to all filter coefficients. This is done to make the signed filter coefficients (supplied through the include file) unsigned to utilize the unsigned multiplier available in the PIC18 family of microcontrollers.

FIR Filter Code

The code for the FIR filter is written in several individual macros. This enables the user to implement the FIR filter in a modular fashion. Flow Charts of the main routine and Interrupt Service Routine are shown in Figure E-1 and Figure E-2, respectively.

The example code (`expl_fir.asm`) in Appendix D shows how to use the macros to implement an FIR filter. This code example includes several include files and macros. Table 1 lists the include files used and their descriptions.

TABLE 1: FIR FILTER EXAMPLE CODE INCLUDE FILES

File Name	Description
Coef.inc	Defines the number of taps and filter coefficients.
Port.inc	Finds the TRIS ports corresponding to the ports OUT_PORT_HIGH and OUT_PORT_LOW selected by the user, and defines constants for Timer1, CCP2, and A/D Converter initialization.
fir_buf.inc	Defines buffer spaces used by FIR filter macros.
fir_mac.inc	Contains FIR filter macros.
Peri.inc	Contains macros to initialize TRIS ports, Timer1 registers, CCP2 registers, T3CON and A/D Converter registers.
int.inc	Contains a macro that enables interrupt priority, assigns high priority for A/D interrupt, enables high priority interrupt, and enables A/D interrupt.

Table 2 provides a list of the macros used and their descriptions.

TABLE 2: FIR FILTER MACROS

Macro Name	Argument	Other Macros Invoked	Description
FIR_FILTER	None	RPT_MULACC, MULACC	Implements FIR filter. The number of taps and filter coefficients are defined in the coef.inc file.
MULACC	None	None	Multiplies the sample value pointed by FSR0 with the filter coefficient pointed by FSR2. The product available in PRODH:PRODL register is then added to the 24-bit value stored in the output_most, output_middle, and output_least variables.
RPT_MULACC	Rpt, loop	MULACC	Adds instructions to form a loop in which code for the macro MULACC is added 'rpt' times.
INIT_PERIPHERALS	None	None	Sets up/initializes input port, output port, A/D Converter, CCP module and Timer1.
SET_INTR_FILTER	None	None	Sets up interrupt for real-time operation of the filter.
INIT_FILTER	None	None	Initializes the buffers used by the filter at the beginning of the program.

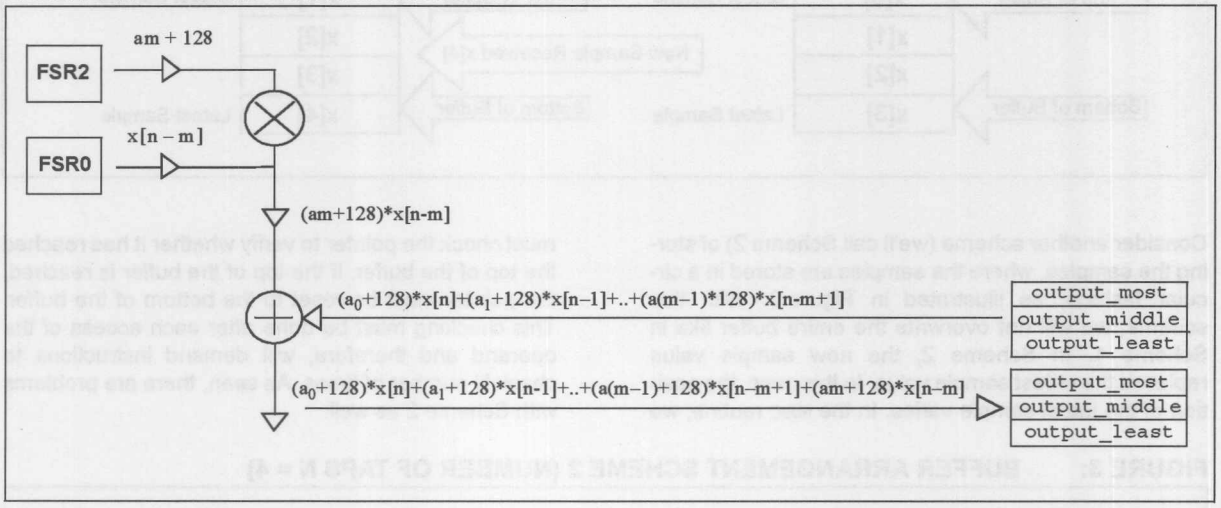
Depending upon the user setup, the parameters listed in Table 3 may need to be assigned.

TABLE 3: FIR FILTER PARAMETERS

Value/Parameter Name	Description/Assignment
IN_PORT	Assign the port used to sample analog signal.
INPUT	The source register of I/P samples to the filter. When the A/D Converter is used, assign ADRESH.
OUT_PORT_HIGH	The port used to output the Most Significant Byte of the filter output. User must assign the port used for this purpose.
OUT_PORT_LOW	The port used to output the Least Significant Byte of the filter output. User must assign the port used for this purpose.
clock_freq	Assign the processor clock frequency used in Hz.
sample_freq	Assign the desired sample frequency in Hz.
num_of_mulacc	Depending upon the sampling frequency required and program memory available, assign a value >= 1 & <= num_of_taps.

Data Storage and Computation

FIGURE 1: COMPUTATION IN MACRO MULACC



After designing the filter, the filter coefficients are to be scaled to integers between -128 and +127. The filter coefficients and length of the filter are entered in the include file before assembling and compilation.

RAM Locations

The following lists the RAM locations used by the software to implement the FIR filter.

- `coeff`

  This RAM area stores 'offset coefficients' of the filter. Offset coefficients are the values obtained by adding 128 to the filter coefficients supplied to the code through the include file. Filter coefficients decide the characteristics of the filter. The user must design the filter coefficients for the required characteristics of the filter.

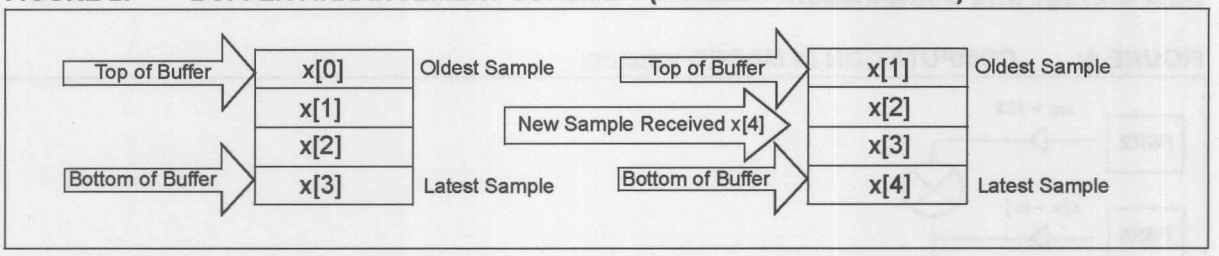
- `buffer`

  This RAM area contains two buffer spaces, `buf1` and `buf0`, which store identical values. The stored values are the sample values of the analog input signal, starting from latest to previous (N–1) values. These values are unsigned 8-bit numbers. Figure 4 illustrates how input samples are stored in the buffers `buf0` and `buf1` for a tap length N equal to 4. The `buf0` buffer space is pointed by FSR0 and `buf1` is pointed by FSR1. The purpose of having two identical buffers is to reduce the number of instructions required to compute the output sample.

Computation requires N multiplications and N accumulations or additions (called as `MAC`) while calculating $y^1[n]$. During each multiplication, different operands are required (one sample value from RAM area `buffer` and one filter coefficient from RAM area `coeff`). Therefore, the pointer should be updated after each multiplication to point to the correct operand. Multiplication is to be done in such a way that the coefficient a_0 should always be multiplied by the latest sample, a_1 should always be multiplied by one sample previous to the latest sample, and so on up to the last coefficient, a_{N-1}, which is to be multiplied by the oldest sample.

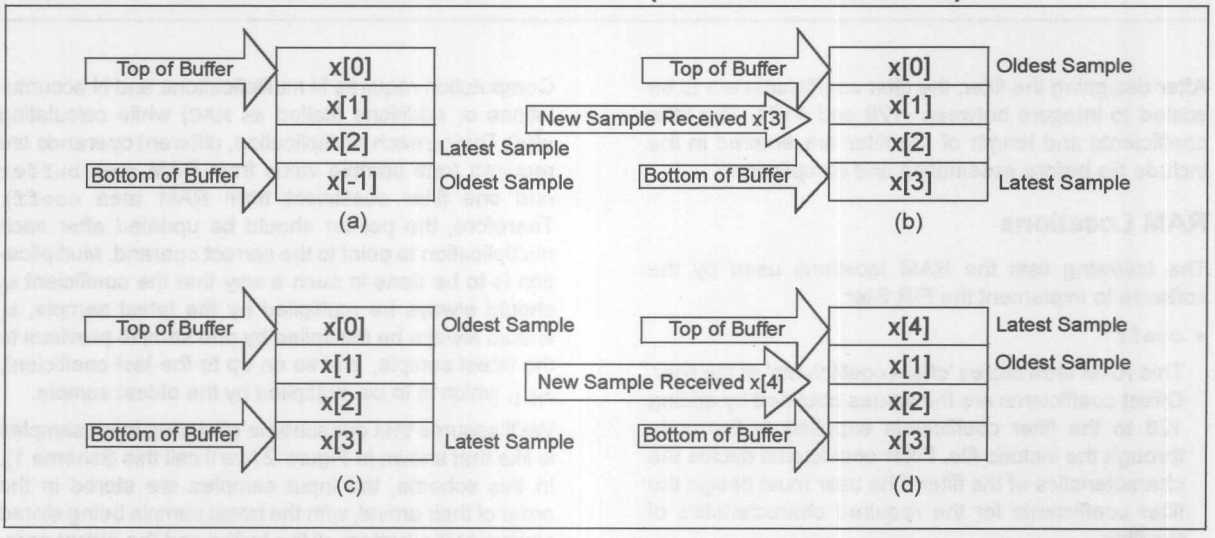
We'll assume that our scheme of storing input samples is like that shown in Figure 2 (we'll call this Scheme 1). In this scheme, the input samples are stored in the order of their arrival, with the latest sample being stored always at the bottom of the buffer and the oldest sample at the top of the buffer. While entering the `MAC` routine, it is sufficient to set the pointer to point to the latest sample. While in `MAC`, it is required to decrement the pointer after each multiplication to be ready to fetch the next operand. Since we are using the FSR register as a pointer, we do not need to add any extra instruction to decrement the pointer to point to the next operand after each multiplication. However, the problem with this type of storing scheme is that after each new sample arrives, the entire buffer must be rewritten N times, as shown in Figure 2. This step adds extra instructions and therefore, increases the computation time.

FIGURE 2: BUFFER ARRANGEMENT SCHEME 1 (NUMBER OF TAPS N = 4)



Consider another scheme (we'll call Scheme 2) of storing the samples, where the samples are stored in a circular fashion, as illustrated in Figure 3. With this scheme, we will not overwrite the entire buffer like in Scheme 1. In Scheme 2, the new sample value replaces the oldest sample value. In this case, the position of the latest sample varies. In the MAC routine, we must check the pointer to verify whether it has reached the top of the buffer. If the top of the buffer is reached, the pointer must be reset to the bottom of the buffer. This checking must be done after each access of the operand and therefore, will demand instructions to check N number of times. As seen, there are problems with Scheme 2 as well.

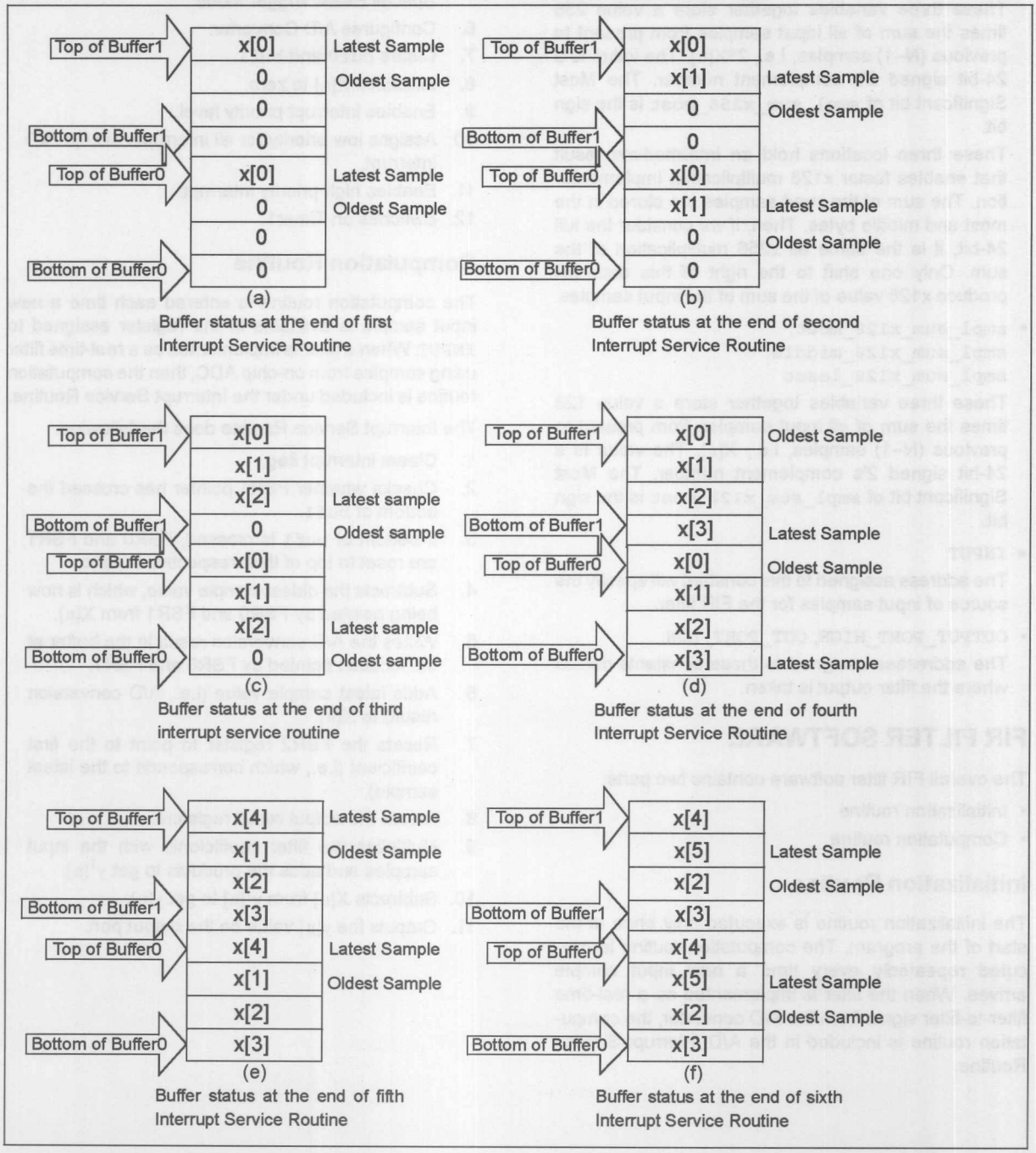
FIGURE 3: BUFFER ARRANGEMENT SCHEME 2 (NUMBER OF TAPS N = 4)



Now consider the scheme illustrated in Figure 4. This scheme stores the samples in the same fashion as Scheme 2, but we have two buffers called `buf0` and `buf1`. After arrival of a new sample, it is stored in both the buffers. At the beginning of the `MAC` routine, we initialize the pointer to point to the latest sample in `buf0`.

Depending upon the latest sample position, at some point of time the pointer will cross the boundary of `buf0`, with the exception of when the latest sample position is at the bottom of `buf0`. In this case, the pointer will not cross the boundary of `buf0`. However, we do not need to check and reset the pointer because it will be pointing to the desired sample, even though the sample is not in `buf0`. Therefore, we have avoided the extra instructions needed to check and reset the pointer, in addition to avoiding adding extra instructions to rewrite the entire buffer after the arrival of each new sample.

FIGURE 4: BUFFER ARRANGEMENT USED IN FIR IMPLEMENTATION



Variables

- output_least, output_middle, output_most

  These three variables together store the computed filter output sample value y[n]. This value is a 24-bit signed 2's complement number. The Most Significant bit of output_most is the sign bit.

- smpl_sum_x256_most, smpl_sum_x256_middle, smpl_sum_x256_least

  These three variables together store a value 256 times the sum of all input samples from present to previous (N–1) samples, i.e., 2*X[n]. The value is a 24-bit signed 2's complement number. The Most Significant bit of smpl_sum_x256_most is the sign bit.

  These three locations hold an intermediate result that enables faster x128 multiplication implementation. The sum of the input samples are stored in the most and middle bytes. Then, if we consider the full 24-bit, it is the same as x256 multiplication of the sum. Only one shift to the right of this data will produce x128 value of the sum of the input samples.

- smpl_sum_x128_most, smpl_sum_x128_middle, smpl_sum_x128_least

  These three variables together store a value 128 times the sum of all input samples from present to previous (N–1) samples, i.e., X[n]. The value is a 24-bit signed 2's complement number. The Most Significant bit of smpl_sum_x128_most is the sign bit.

- INPUT

  The address assigned to this constant will specify the source of input samples for the FIR filter.

- OUTPUT_PORT_HIGH, OUT_PORT_LOW

  The addresses assigned to these constants decide where the filter output is taken.

FIR FILTER SOFTWARE

The overall FIR filter software contains two parts:

- Initialization routine
- Computation routine

Initialization Routine

The initialization routine is executed only once at the start of the program. The computation routine is executed repeatedly every time a new input sample arrives. When the filter is implemented as a real-time filter-to-filter signal from the A/D converter, the computation routine is included in the A/D Interrupt Service Routine.

The Initialization routine does the following:

1. Stores the offset filter coefficients in RAM area coeff. The value of offset filter coefficients is obtained by adding 128 to the filter coefficients entered in the include file.
2. Configures OUTPUT_PORT_HIGH and OUTPUT_PORT_LOW as output ports.
3. Configures IN_PORT.
4. Clears TMR1H:TMR1L registers.
5. Configures CCP2 module for compare in Special Event Trigger mode.
6. Configures A/D Converter.
7. Clears buf0 and buf1.
8. Initializes X[n] to zero.
9. Enables interrupt priority level.
10. Assigns low priority for all interrupts except A/D interrupt.
11. Enables high priority interrupt.
12. Switches on Timer1.

Computation Routine

The computation routine is entered each time a new input sample is available in the register assigned to INPUT. When a filter is implemented as a real-time filter using samples from on-chip ADC, then the computation routine is included under the Interrupt Service Routine.

The Interrupt Service Routine does the following:

1. Clears interrupt flag.
2. Checks whether FSR1 pointer has crossed the bottom of buf1.
3. If bottom of buf1 is crossed, FSR0 and FSR1 are reset to top of their respective buffers.
4. Subtracts the oldest sample value, which is now being pointed by FSR0 and FSR1 from X[n].
5. Writes the A/D conversion result in the buffer at the location pointed by FSR0 and FSR1.
6. Adds latest sample value (i.e., A/D conversion result) to X[n].
7. Resets the FSR2 register to point to the first coefficient (i.e., which corresponds to the latest sample).
8. Clears the output result registers.
9. Multiplies the filter coefficients with the input samples and adds the products to get $y^1[n]$.
10. Subtracts X[n] from $y^1[n]$ to get y[n].
11. Outputs the y[n] value on the output port.

The time spent for the execution of the Interrupt Service Routine limits the maximum sampling frequency. The code provides a trade-off between execution time and the amount of program memory used by means of the value entered for the constant num_of_mulacc. This value gives the number of MAC routines used in the software loop in the Interrupt Service Routine. The higher the value for this constant, the lower the execution time, which enables us to go to a higher sampling frequency. For a filter of tap length N, the value of num_of_mulacc can range from 1 to N. The user should ensure that for the entered value of num_of_mulacc, there is a sufficient number of available program memory locations.

Code Examples

Following are two examples of code for num_of_taps=31, and num_of_mulacc=1 and num_of_mulacc=2. These examples illustrate how the code changes with num_of_mulacc. The code in italics forms the MAC routine. Num_of_mulacc specifies how many times the MAC routine is repeated in a loop. In Example 1, the instruction decfsz count, F is executed 31 times, and the instruction goto Loop1 is executed 30 times. In Example 2, the same instructions are executed 15 and 14 times, respectively. Therefore, Example 2 takes less time for computation than Example 1. However, Example 2 requires more program memory than Example 1.

EXAMPLE 1: num_of_mulacc=1

```
        movlw   loop
        movwf   count   ;Loop = D'31'
Loop1   movf    POSTDEC0,W
        mulwf   POSTINC2
        movf    PRODL,W
        addwf   output_least
        movf    PRODH,W
        addwfc  output_middle
        clrf    WREG
        addwfc  output_most
        decfsz  count, F
        goto    Loop1
```

(This loop is executed 31 times)

EXAMPLE 2: num_of_mulacc=2

```
        movlw   loop
        movwf   count       ;Loop = D'15'
Loop1   movf    POSTDEC0,W
        mulwf   POSTINC2
        movf    PRODL,W
        addwf   output_least
        movf    PRODH,W
        addwfc  output_middle
        clrf    WREG
        addwfc  output_most
        movf    POSTDEC0,W
        mulwf   POSTINC2
        movf    PRODL,W
        addwf   output_least
        movf    PRODH,W
        addwfc  output_middle
        clrf    WREG
        addwfc  output_most
        decfsz  count, F
        goto    Loop1
        movf    POSTDEC0,W
        mulwf   POSTINC2
        movf    PRODL,W
        addwf   output_least
        movf    PRODH,W
        addwfc  output_middle
        clrf    WREG
        addwfc  output_most
```

(This loop is executed 15 times)

Procedure to Implement an FIR Filter

This procedure references freeware (see Appendix F) used to generate coefficients.

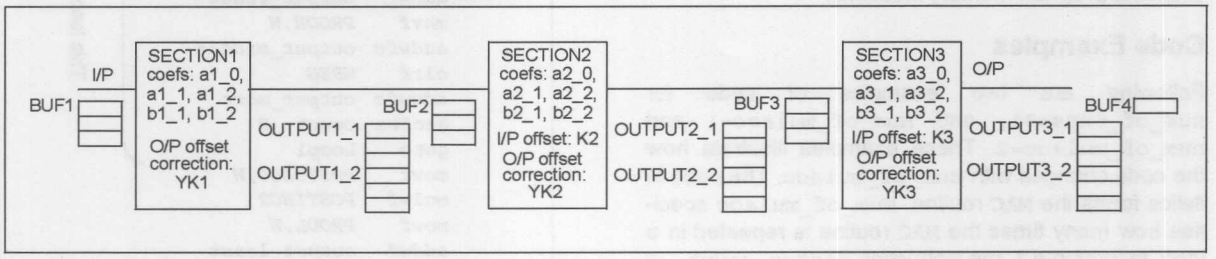
1. Determine the maximum frequency, for example, F Hz of the signal to be filtered.
2. Choose a sampling frequency (Fs ≥ 2F Hz).
3. Decide on the filter characteristics required.
4. Input the filter characteristics using the coefficient generation freeware to get the coefficients.
5. Scale the coefficients so they are integers between -128 and +127.
6. Add the scaled coefficients and the number of taps into the include file.
7. Build and generate the HEX code.
8. Transfer the program to the PIC18 microcontroller.
9. Run the program and check the filter characteristics.

Note: Steps 1 through 6 are explained in greater detail in Appendix B.

IIR FILTER IMPLEMENTATION

The IIR filter is implemented in the form of a number of sections connected in cascade, as shown in Figure 5. Each section is referred to as a BIQUAD section. Each BIQUAD section itself is an IIR filter, which computes output samples from the present input sample, two previous input samples and two previous output samples. Implementing the overall IIR filter in the form of BIQUAD sections decreases the sensitivity to round-off errors and gives better control to ensure stability of the filter.

FIGURE 5: IMPLEMENTATION OF IIR FILTER IN THE FORM OF BIQUAD SECTIONS CONNECTED IN CASCADE



Each BIQUAD section implements the following equation.

EQUATION 4: BIQUAD EQUATION

$$yi[n] = ai\_0*xi[n] + ai\_1*xi[n-1] + ai\_2*xi[n-2] - bi\_1*yi[n-1] - bi\_2*yi[n-2]$$

Where xi[n] denotes the nth input sample, yi[n] denotes nth output sample of section i and ai_0, ai_1, ai_2, bi_1 and bi_2 are the filter coefficients of section i.

The code for the IIR filter is written in the form of several macros. This enables the user to implement the IIR filter in a modular fashion. Flow charts of the main routine and Interrupt Service Routine are shown in Figure E-3 and Figure E-4, respectively.

The example code (expl_iir.asm) in Appendix D shows how to use the macros to implement an IIR filter. This code example includes several include files and macros. Table 4 lists the include files used and their descriptions:

TABLE 4: IIR FILTER INCLUDE FILES.

File Name	Description
Coef.inc	Defines the filter characteristics. This file defines filter coefficients of each BIQUAD section, the number of BIQUAD sections used, input offset constants K2, K3, etc., and output offset correction YK1,YK2, YK3, etc.
Port.inc	Determines the TRIS ports corresponding to the ports OUT_PORT_HIGH and OUT_PORT_LOW selected by the user. Defines constants for Timer1, CCP2, and A/D Converter initialization.
iir_buf.inc	Defines buffer spaces used by IIR filter macros.
iir_mac.inc	Contains macros of the IIR filter.
Peri.inc	Contains macros to initialize TRIS ports, Timer1 registers, CCP2 registers, T3CON and A/D Converter registers.
int.inc	Contains the macro which enables interrupt priority, assigns high priority for A/D interrupt, enables high priority interrupt, and enables A/D interrupt.

Table 5 provides a list of macros used and their descriptions:

TABLE 5: IIR FILTER MACROS

Macro Name	Arguments	Other Macros Invoked	Description
IIR_FILTER	None	BIQUAD, TRNSFR, UNSIGNXSIGN_0, UNSIGNXSIGN, SIGNXSIGN, CLEAR	Implements an IIR filter in the form of BIQUAD sections connected in cascade. The number of BIQUAD sections used and the coefficients for each section are input from the include file.
BIQUAD	Input, a0, a1, a2, b1, b2, output, output1, output2	TRNSFR, UNSIGNXSIGN_0, UNSIGNXSIGN, SIGNXSIGN, CLEAR	This macro implements one BIQUAD IIR filter section by implementing the equation: y[n]=a0*x[n]+a1*x[n–1]+a2*x[n–2]-b1*y[n–1]-b2*y[n–2] where x's and y's refer to input and output values of the BIQUAD IIR filter and a0, a1, a2, b1, and b2 are filter coefficients.
UNSIGNXSIGN_0	X, coef, acc	CLEAR	Multiplies unsigned value in register X with the signed literal value coef. The result is spread over the locations acc, acc+1, acc+2, and acc+3. This macro is intended to be used at the beginning of a series of multiply accumulate operations.
UNSIGNXSIGN	X, coef, acc	None	Multiplies unsigned value in register X with signed literal value 'coef'. The result is spread over the locations acc, acc+1, acc+2 and acc+3. This macro is intended to be used after using the macro UNSIGNXSIGN_0 at the beginning of a series of multiply accumulate operations.
SIGNXSIGN	X, coef, acc	None	Multiplies the signed value stored at locations X, X+1 and X+2 with the signed value supplied through literal constant coef. The product is subtracted from the value stored in locations acc, acc+1,acc+2 and acc+3.
CLEAR	loc	None	Clears consecutive three locations loc, loc+1 and loc+2.
INIT_PERIPHERALS	None	None	Sets up/initializes input port, output port, A/D Converter, CCP module and Timer1.
SET_INTR_FILTER	None	None	Sets up interrupt for real-time operation of the filter.
INIT_FILTER	None	CLEAR	Initializes the buffers used by the filter at the beginning of the program.

Depending upon the user setup, the parameters listed in Table 6 may need to be assigned.

TABLE 6: IIR FILTER PARAMETERS

Value/Parameter Name	Description/Assignment
IN_PORT	The port used to sample the analog signal.
INPUT	The source register of I/P samples to the filter. When the A/D Converter is used to assign ADRESH.
OUT_PORT_HIGH	The port used to output the Most Significant Byte of the filter output. User must assign the port used for this purpose.
OUT_PORT_LOW	The port used to output the Least Significant Byte of the filter output. User must assign the port used for this purpose.
clock_freq	Assign the processor clock frequency used in Hz.
sample_freq	Assign the desired sample frequency in Hz.

Data Storage and Computation

Figure 6 shows the input and output buffers used for each `BIQUAD` section.

FIGURE 6: COMPUTATION IN MACRO BIQUAD

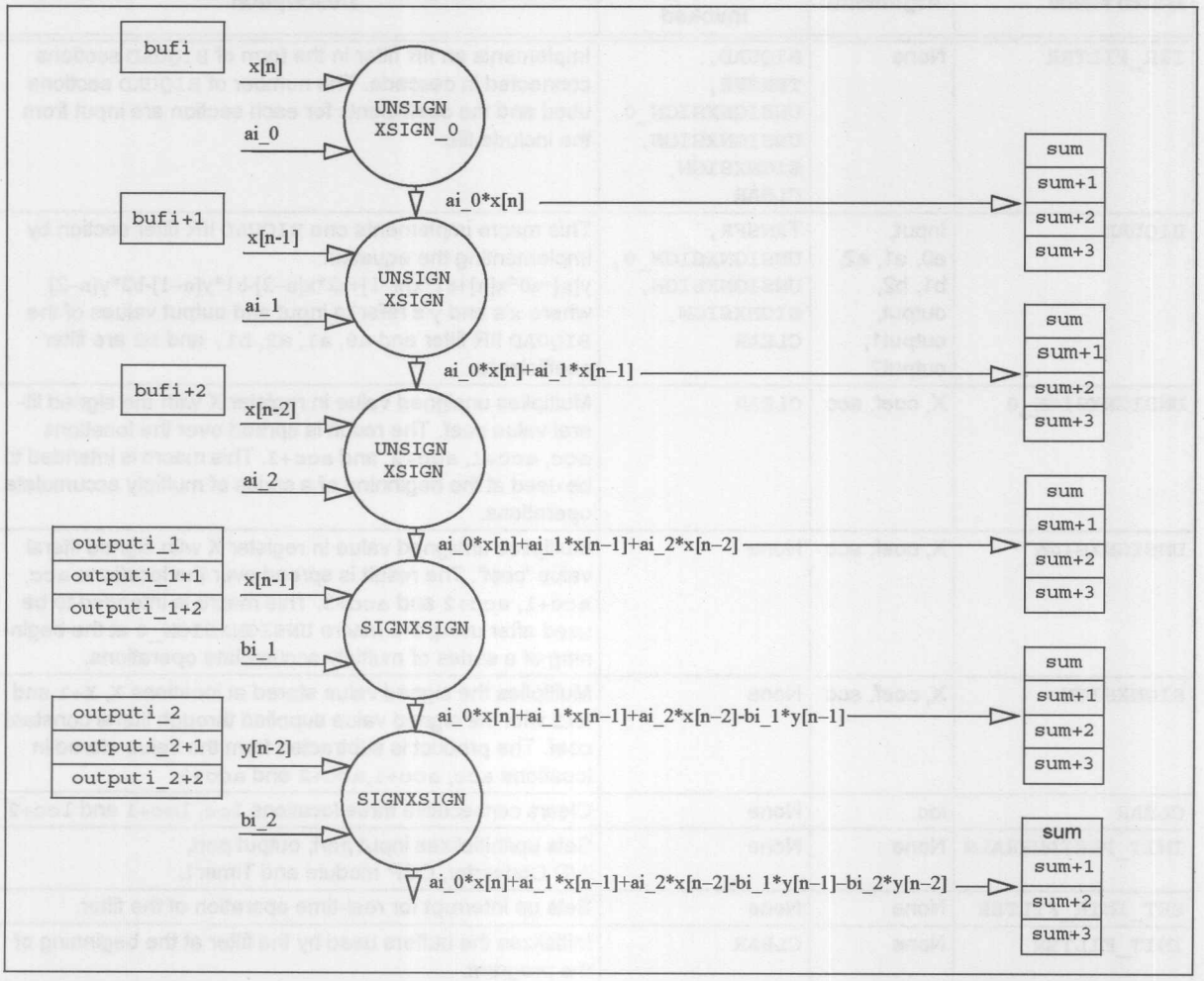
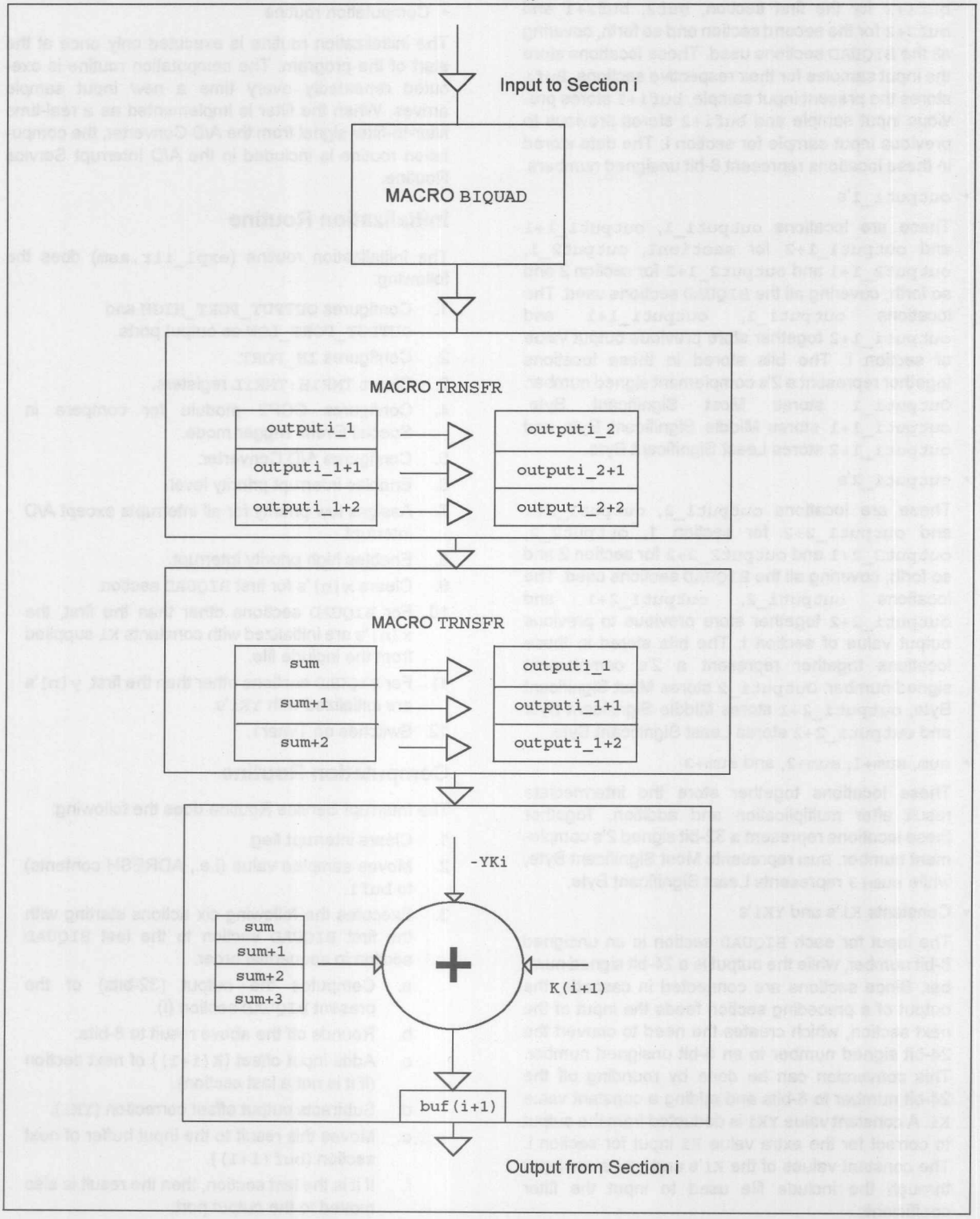


Figure 7 shows the computations performed to compute the output.

FIGURE 7: COMPUTATION PERFORMED IN A BIQUAD SECTION i



Memory Locations

- bufi's

  These are memory locations buf1, buf1+1 and buf1+2 for the first section, buf2, buf2+1 and buf2+2 for the second section and so forth, covering all the BIQUAD sections used. These locations store the input samples for their respective sections. Bufi stores the present input sample, bufi+1 stores previous input sample and bufi+2 stores previous to previous input sample for section i. The data stored in these locations represent 8-bit unsigned numbers.

- outputi_1's

  These are locations output1_1, output1_1+1 and output1_1+2 for section1, output2_1, output2_1+1 and output2_1+2 for section 2 and so forth, covering all the BIQUAD sections used. The locations outputi_1, outputi_1+1 and outputi_1+2 together store previous output value of section i. The bits stored in these locations together represent a 2's complement signed number. Outputi_1 stores Most Significant Byte, outputi_1+1 stores Middle Significant Byte and outputi_1+2 stores Least Significant Byte.

- outputi_2's

  These are locations output1_2, output1_2+1 and output1_2+2 for section 1, output2_2, output2_2+1 and output2_2+2 for section 2 and so forth, covering all the BIQUAD sections used. The locations outputi_2, outputi_2+1 and outputi_2+2 together store previous to previous output value of section i. The bits stored in these locations together represent a 2's complement signed number. Outputi_2 stores Most Significant Byte, outputi_2+1 stores Middle Significant Byte and outputi_2+2 stores Least Significant Byte.

- sum, sum+1, sum+2, and sum+3

  These locations together store the intermediate result after multiplication and addition. Together these locations represent a 32-bit signed 2's complement number. Sum represents Most Significant Byte, while sum+3 represents Least Significant Byte.

- Constants Ki's and YKi's

  The input for each BIQUAD section is an unsigned 8-bit number, while the output is a 24-bit signed number. Since sections are connected in cascade, the output of a preceding section feeds the input of the next section, which creates the need to convert the 24-bit signed number to an 8-bit unsigned number. This conversion can be done by rounding off the 24-bit number to 8-bits and adding a constant value Ki. A constant value YKi is deducted from the output to correct for the extra value Ki input for section i. The constant values of the Ki's and YKi's are input through the include file used to input the filter coefficients.

IIR Filter Software

The overall IIR filter software contains two parts:

- Initialization routine
- Computation routine

The initialization routine is executed only once at the start of the program. The computation routine is executed repeatedly every time a new input sample arrives. When the filter is implemented as a real-time filter-to-filter signal from the A/D Converter, the computation routine is included in the A/D Interrupt Service Routine.

Initialization Routine

The initialization routine (expl_iir.asm) does the following:

1. Configures OUTPUT_PORT_HIGH and OUTPUT_PORT_LOW as output ports.
2. Configures IN_PORT.
3. Clears TMR1H:TMR1L registers.
4. Configures CCP2 module for compare in Special Event Trigger mode.
5. Configures A/D Converter.
6. Enables interrupt priority level.
7. Assigns low priority for all interrupts except A/D interrupt.
8. Enables high priority interrupt.
9. Clears x[n]'s for first BIQUAD section.
10. For BIQUAD sections other than the first, the x[n]'s are initialized with constants Ki supplied from the include file.
11. For BIQUAD sections other than the first, y[n]'s are initialized with YKi's.
12. Switches on Timer1.

Computation Routine

The Interrupt Service Routine does the following:

1. Clears interrupt flag.
2. Moves sampled value (i.e., ADRESH contents) to buf1.
3. Executes the following six actions starting with the first BIQUAD section to the last BIQUAD section in sequential order.
   a. Computes the output (32-bits) of the present BIQUAD section (i).
   b. Rounds off the above result to 8-bits.
   c. Adds input offset (K(i+1)) of next section (if it is not a last section).
   d. Subtracts output offset correction (YKi).
   e. Moves this result to the input buffer of next section (buf(i+1)).
   f. If it is the last section, then the result is also moved to the output port.

Procedure to Implement an IIR filter

This procedure requires the use of digital filter coefficient generation freeware (see Appendix F) and a Microsoft® Excel® spreadsheet 'coef modifier' (see Figure C-4). This spreadsheet is available for download from the Microchip web site (see Appendix G for more information).

1. Determine the maximum frequency (e.g., F Hz) of the signal to be filtered.
2. Choose a sampling frequency Fs ≥ 2F Hz.
3. Decide on the filter characteristics required and arrive at filter specifications.
4. Input filter specifications using digital filter coefficient generation freeware to get the coefficients `a0`, `a1`, `a2`, `b1` and `b2` for each `BIQUAD` section and the number of `BIQUAD` sections required.
5. Determine the maximum gain 'gc' for each `BIQUAD` section.
6. Enter the coefficients (`a1_0`, `a1_1`, `a1_2`, `b1_1`...) and gain (`gc1`, `gc2`, ...) into the spreadsheet to get the gain normalized coefficients.
7. Determine the optimum input offset constants `Ki`'s for each `BIQUAD` section other than the first `BIQUAD` section and output offset correction from the spreadsheet.
8. Enter the modified coefficients, input offset coefficients `Ki`'s, and output offset correction constants `YKi`'s into the include file `coef.inc`.
9. Build and generate the HEX code.
10. Transfer the program to the PIC18 microcontroller.
11. Run the program and check the filter characteristics.

Note: Steps 1 through 8 are explained in greater detail in Appendix C.

TESTING AND PERFORMANCE

The FIR and IIR filter can be used for off-line processing or for real-time processing. The code examples, `expl_fir.asm` (for FIR) and `expl_iir.asm` (for IIR) in Appendix D, demonstrate how to filter an analog signal input to one of the I/P ports of an on-chip A/D Converter, and get the filtered 2 bytes output through any of the ports assigned to `OUT_PORT_HIGH` and `OUT_PORT_LOW`. The block diagram of this setup is shown in Figure 8.

The FIR and IIR filters were tested on a PICDEM™ 2 demo board using a PIC18C452 microcontroller (see Figure 9 for the circuit diagram). The analog signal to be filtered is fed through PORTA pin AN1. The code performs computation on the input samples x[n] and generates one output sample y[n] each time one sample is input through an A/D Interrupt Service Routine.

The filtered output samples y[n] are fed to the 16-bit DAC constructed with PORTB (LS Byte) and PORTD (MS Byte) outputs and the R-2R ladder network on the PICDEM 2 demo board. Between each input (DA0-DA15) and output (DA-OP), the R-2R ladder network can be viewed as a voltage divider, which gives a fraction of the input voltage at the output. The output voltage is the sum of contributions of all the sixteen inputs.

The resistor values are such that the contribution at the output due to each input is proportional to their bit weightage. Therefore, the Least Significant bit (DA0) gives the least contribution at the output and the Most Significant bit (DA15) contributes highest and 2^{15} times more than the contribution of the Least Significant bit. Equivalently, the output is the weighed sum of the input bits. Therefore, it represents the equivalent analog value of the digital word at PORTB and PORTD outputs. Please note that a logic '1' at PORTD7 produces a -5V at Q1 collector (because transistor Q1 switches off), unlike at other PORTD and PORTB pins, which produce a 5V for logic '1'. This is because of the 2's complement notation used to represent the numbers.

The -5V for the transistor Q1 is to be supplied from an external source because there is no -5V supply available in the PICDEM 2 demo board.

The output from the DAC is then passed through a low-pass filter, whose cut-off frequency is half the sampling frequency. The final analog output is available at the output of this low-pass filter.

FIGURE 8: TEST SETUP OF FIR/IIR FILTER

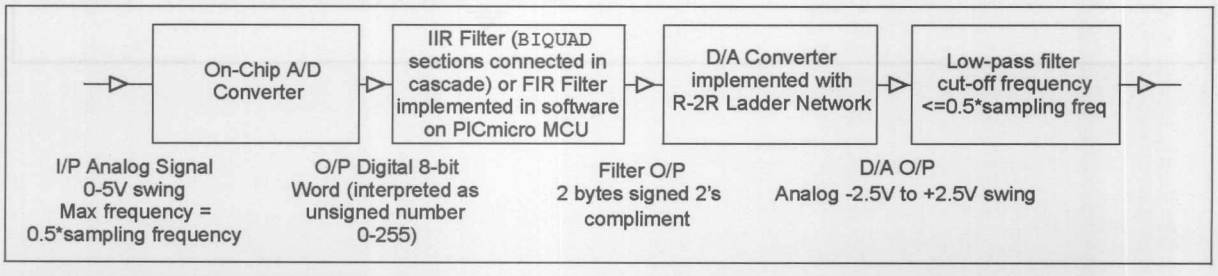
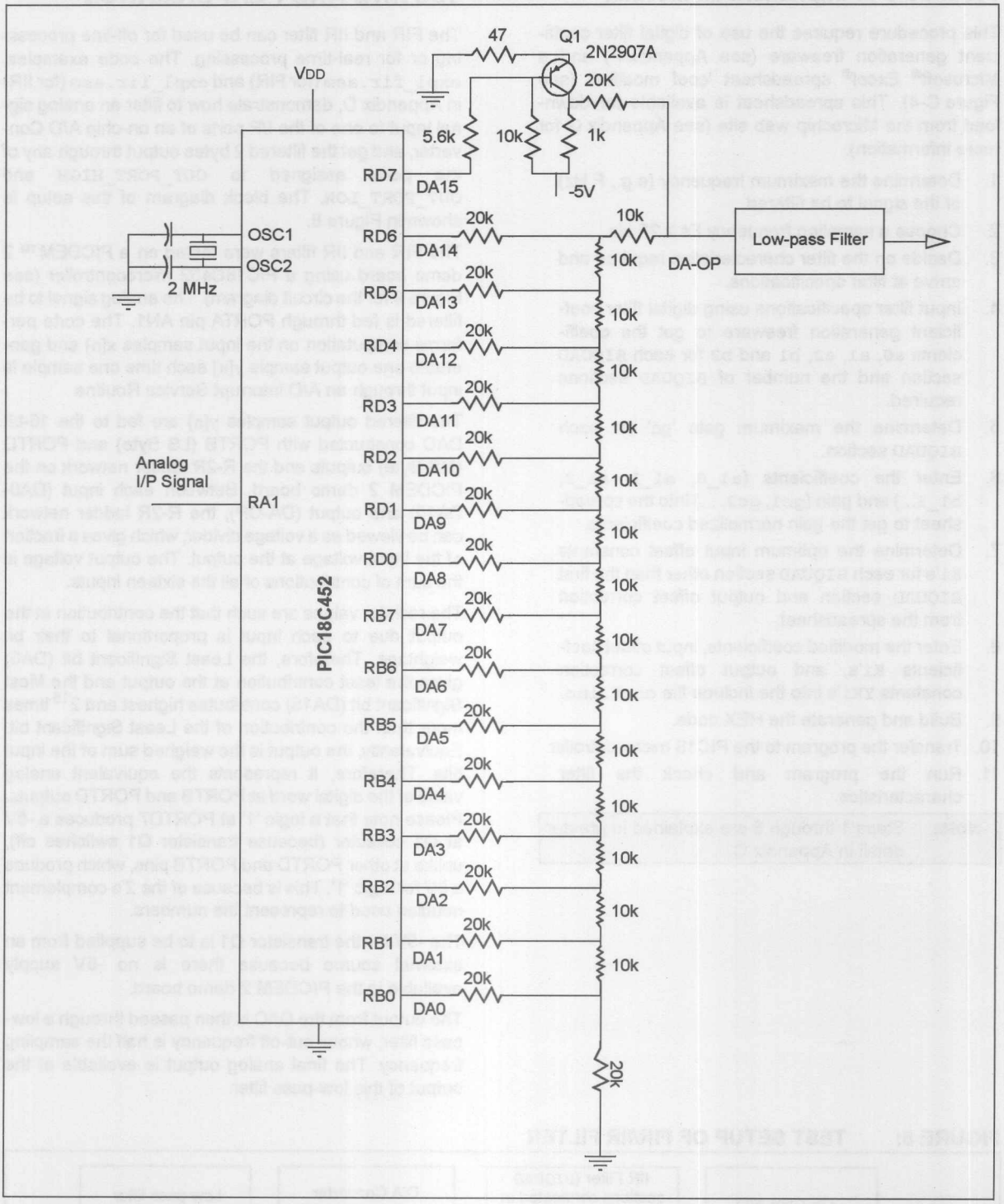


FIGURE 9: CIRCUIT DIAGRAM FOR IMPLEMENTATION OF FIR AND IIR FILTERS



Sampling and A/D Conversion

The analog signal to be filtered is input to the on-chip PIC18C452 A/D Converter through PORTA pin AN1. The A/D Converter outputs a digital number representing the analog signal level in 8-bit unsigned format.

To sample the input analog signal at regular intervals, CCP2 is used in Compare mode with a special event trigger feature. The processor clock frequency is to be entered for the value of `clock_freq` in the `expl_fir.asm` or `expl_iir.asm` files. The sampling rate of the input analog signal is determined by the value entered for `sample_freq` in either the `expl_fir.asm` or the `expl_iir.asm` files. The literal values `comph` and `compl` are computed (automatically during compilation time) using sampling frequency `sample_freq` and clock frequency `clock_freq` and then loaded to registers `CCPR2H` and `CCPR2L`, respectively.

The sampling rate of the input analog signal is controlled by the value in the `CCPR2H:CCPR2L` register. The CCP2 module that uses these registers is configured to work as a compare module in Special Event Trigger mode. In Compare mode, the 16-bit `CCPR2H:CCPR2L` value is constantly compared against the `TMR1H:TMR1L` value. TMR1 is configured to work on the processor internal clock. When a match occurs, an internal hardware trigger is generated. This trigger resets the `TMR1H:TMR1L` register and starts A/D conversion. This Trigger mode is known as 'Special Event Trigger mode'.

Following are the advantages of using the CCP module in Special Event Trigger mode.

1. Accurate sampling interval is maintained.
2. `TMR1H:TMR1L` is cleared automatically after overflow.
3. GO bit is set automatically after `TMR1H:TMR1L` overflow.
4. Had we used Timer1 without CCP module, 2 interrupt routines would have been required: one for Timer1 overflow and the other for A/D interrupt.
5. Because of the previous reasons, code length is reduced which is very critical for signal processing.

FIR Filter Performance

Filter coefficients were designed for a low-pass and a high-pass filter, each of tap length 31. The coefficients were input in the include file and the performance was checked. The frequency response of these filters are shown in Figure 10 and Figure 11, respectively. The corresponding include files are shown in Example 3 and Example 4.

FIGURE 10: FREQUENCY RESPONSE OF LOW-PASS FILTER 500 Hz CUT-OFF

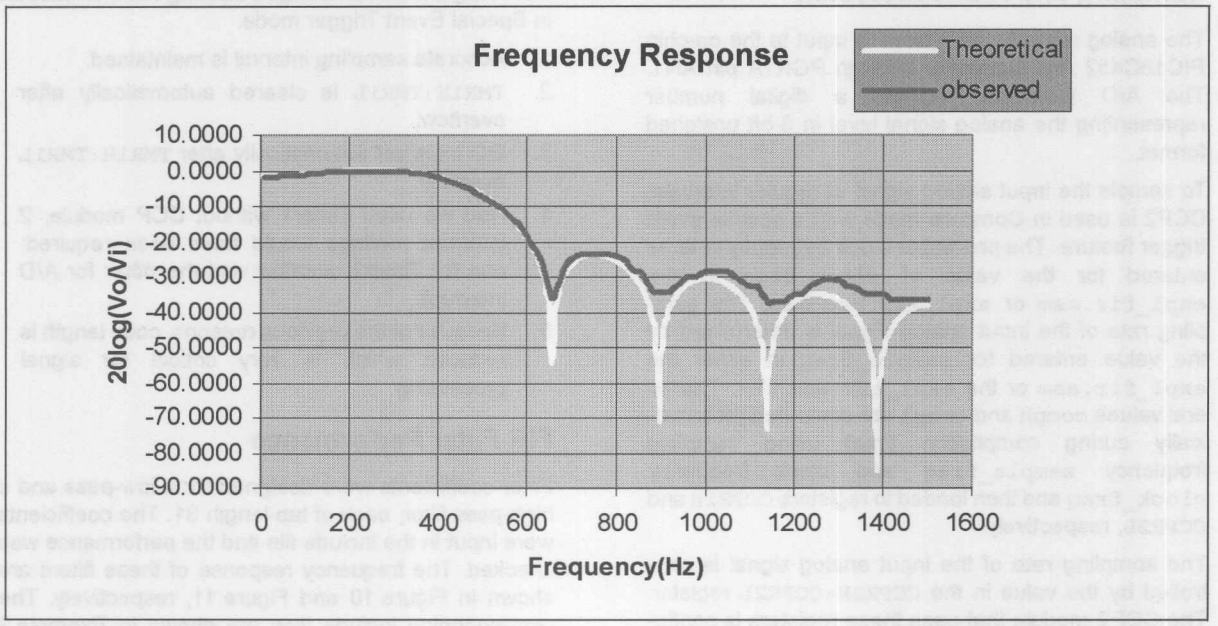
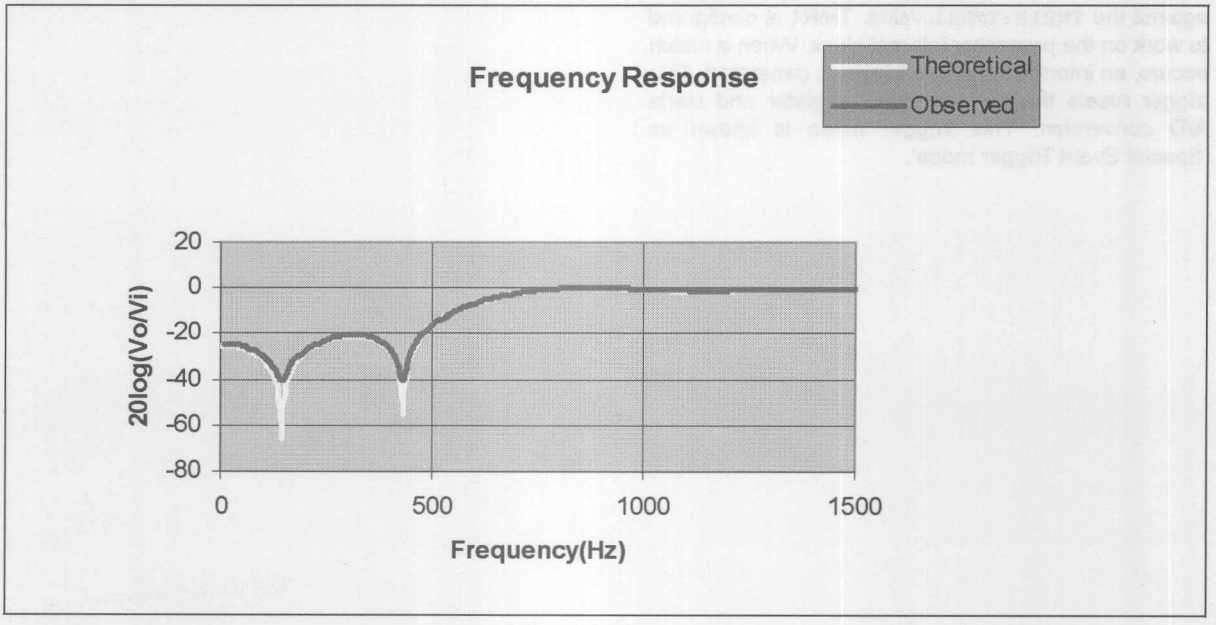


FIGURE 11: FREQUENCY RESPONSE OF HIGH-PASS FILTER 600 Hz CUT-OFF



EXAMPLE 3: LOW-PASS FILTER (500 Hz CUT-OFF) INCLUDE FILE

```
;*******************************************************************************
;Low pass filter
;Sampling frequency 8000 Hz
;Number of taps 31
;Pass band ripple 1 dB
;Stop band attenuation 40dB
;Cut-off frequency 500 Hz
;Stop band frequency 600 Hz
;*******************************************************************************

 CONSTANT num_of_taps=D'31'          ;Enter the filter tap length here
 CONSTANT dist_to_last_tap=num_of_taps-1
 CONSTANT dist_to_prv_to_last_tap=num_of_taps-2
;define filter coeffs here
       CONSTANT coeff0=0xf8          ;corresponds to the latest sample
       CONSTANT coeff1=0xf0
       CONSTANT coeff2=0xe9
       CONSTANT coeff3=0xe5
       CONSTANT coeff4=0xe5
       CONSTANT coeff5=0xe9
       CONSTANT coeff6=0xf2
       CONSTANT coeff7=0x0
       CONSTANT coeff8=0x12
       CONSTANT coeff9=0x26
       CONSTANT coeff10=0X3c
       CONSTANT coeff11=0X51
       CONSTANT coeff12=0X64
       CONSTANT coeff13=0x72
       CONSTANT coeff14=0x7c
       CONSTANT coeff15=0x7f
       CONSTANT coeff16=0x7c
       CONSTANT coeff17=0x72
       CONSTANT coeff18=0x64
       CONSTANT coeff19=0x51
       CONSTANT coeff20=0x3c
       CONSTANT coeff21=0x26
       CONSTANT coeff22=0x12
       CONSTANT coeff23=0x0
       CONSTANT coeff24=0xf2
       CONSTANT coeff25=0xe9
       CONSTANT coeff26=0xe5
       CONSTANT coeff27=0xe5
       CONSTANT coeff28=0xe9
       CONSTANT coeff29=0xf0
       CONSTANT coeff30=0xf8         ;corresponds to the oldest sample
```

EXAMPLE 4: HIGH-PASS FILTER (600 Hz CUT-OFF) INCLUDE FILE

```
;*******************************************************************************
;High pass filter
;Sampling frequency 8000 Hz
;Number of taps 31
;Pass band ripple 1 dB
;Stop band attenuation 40dB
;Cut-off frequency 600 Hz
;Stop band frequency 500 Hz
;*******************************************************************************

 CONSTANT num_of_taps=D'31'       ;Enter the filter tap length here
 CONSTANT dist_to_last_tap=num_of_taps-1
 CONSTANT dist_to_prv_to_last_tap=num_of_taps-2
;define filter coeffs here
       CONSTANT coeff0=0xfe       ;corresponds to the latest sample
       CONSTANT coeff1=0xff
       CONSTANT coeff2=0x01
       CONSTANT coeff3=0x02
       CONSTANT coeff4=0x04
       CONSTANT coeff5=0x05
       CONSTANT coeff6=0x05
       CONSTANT coeff7=0x03
       CONSTANT coeff8=0x01
       CONSTANT coeff9=0xfe
       CONSTANT coeff10=0Xf9
       CONSTANT coeff11=0Xf5
       CONSTANT coeff12=0Xf0
       CONSTANT coeff13=0xed
       CONSTANT coeff14=0xea
       CONSTANT coeff15=0x7f
       CONSTANT coeff16=0xea
       CONSTANT coeff17=0xed
       CONSTANT coeff18=0xf0
       CONSTANT coeff19=0xf5
       CONSTANT coeff20=0xf9
       CONSTANT coeff21=0xfe
       CONSTANT coeff22=0x01
       CONSTANT coeff23=0x03
       CONSTANT coeff24=0x05
       CONSTANT coeff25=0x05
       CONSTANT coeff26=0x04
       CONSTANT coeff27=0x02
       CONSTANT coeff28=0x01
       CONSTANT coeff29=0xff
       CONSTANT coeff30=0xfe      ;corresponds to the oldest sample
```

The observed execution times and the corresponding maximum sampling frequencies for filter of tap length 31 are shown in Table 7.

TABLE 7: 31 TAP FILTER PERFORMANCE STATISTICS

num_of_mulacc	ISR Execution Time	Maximum Sampling Frequency Possible	MIPs Requirement at 8 kHz Sampling Frequency
1	76.8 µsec	13.02 kHz	3.07
31	58 µsec	17.24 kHz	2.32

Note: Processor clock frequency = 20 MHz.

Table 8 lists the memory requirements for the 31 tap filter.

TABLE 8: 31 TAP FILTER MEMORY REQUIREMENTS

num_of_mulacc	Program Memory Locations	Data Memory Locations
1	358	103
31	828	103

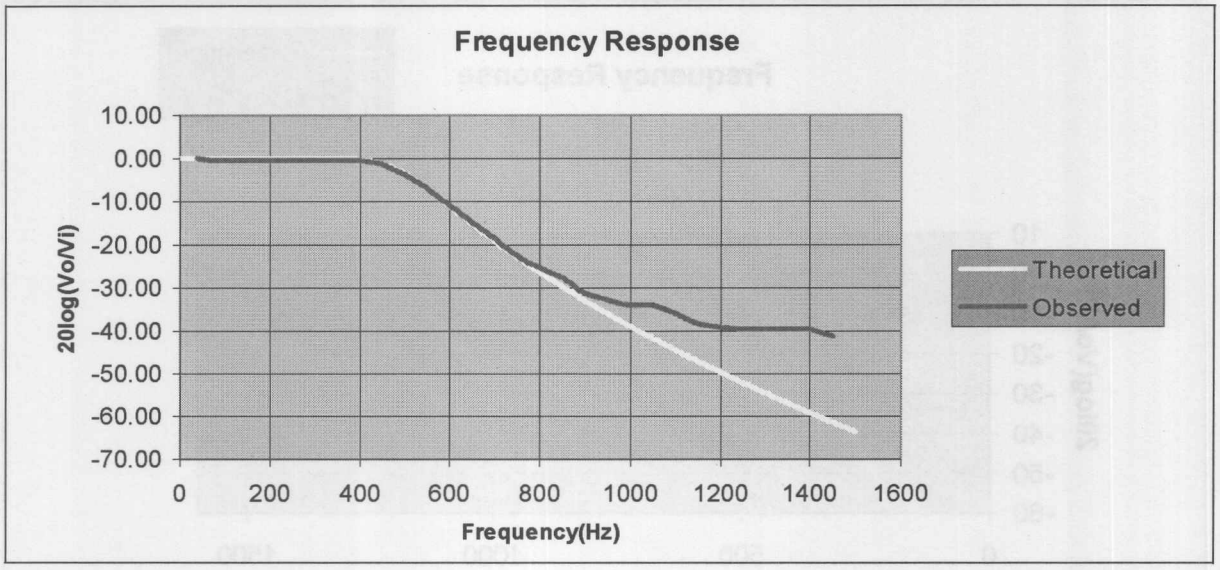
IIR Filter Performance

Filter coefficients were designed for low-pass and high-pass filters with three BIQUAD sections. The following are the filter coefficients include files and response plots of these filters.

EXAMPLE 5: LOW-PASS FILTER INCLUDE FILE

```
;*****************************************************************
;Low pass Butterworth filter
;cut-off frequency 500 Hz
;*****************************************************************

;specify the number of biquad sections in the following line
CONSTANT NUMBER_OF_SECTIONS=3
;******************SECTION 1*************************************
CONSTANT a1_0=0X4
CONSTANT a1_1=0X9
CONSTANT a1_2=0X4

CONSTANT b1_1=0x3B0
CONSTANT b1_2=0XD2
CONSTANT YK1=D'0' ;enter this value in decimal representation only
;******************SECTION 2*************************************
CONSTANT K2=0x40
CONSTANT a2_0=0X7
CONSTANT a2_1=0Xe
CONSTANT a2_2=0X7

CONSTANT b2_1=0X35b
CONSTANT b2_2=0X77
CONSTANT YK2=D'64' ;enter this value in decimal representation only
;******************SECTION 3*************************************
CONSTANT K3=0x40
CONSTANT a3_0=0x7
CONSTANT a3_1=0XF
CONSTANT a3_2=0X7

CONSTANT b3_1=0X376
CONSTANT b3_2=0X94
CONSTANT YK3=D'62' ;enter this value in decimal representation only
;*****************************************************************
```

FIGURE 12: FREQUENCY RESPONSE OF LOW-PASS FILTER 500 Hz CUT-OFF



EXAMPLE 6: HIGH-PASS FILTER INCLUDE FILE

```
;*****************************************************************
;High pass Butterworth filter
;cut-off frequency 600 Hz
;sampling frequency 8000 Hz
;*****************************************************************

;specify the number of biquad sections in the following line
CONSTANT NUMBER_OF_SECTIONS=3
;******************SECTION 1*************************************
CONSTANT a1_0=0X6c
CONSTANT a1_1=0X2d7
CONSTANT a1_2=0X6c

CONSTANT b1_1=0x39b
CONSTANT b1_2=0Xcb
CONSTANT YK1=D'0' ;enter this value in decimal representation only
;******************SECTION 2*************************************
CONSTANT K2=0x80
CONSTANT a2_0=0Xa8
CONSTANT a2_1=0X350
CONSTANT a2_2=0Xa8

CONSTANT b2_1=0X340
CONSTANT b2_2=0X66
CONSTANT YK2=D'0' ;enter this value in decimal representation only
;******************SECTION 3*************************************
CONSTANT K3=0x80
CONSTANT a3_0=0xb6
CONSTANT a3_1=0X36c
CONSTANT a3_2=0Xb6

CONSTANT b3_1=0X35c
CONSTANT b3_2=0X85
CONSTANT YK3=D'0' ;enter this value in decimal representation only
;*****************************************************************
```

FIGURE 13: FREQUENCY RESPONSE OF HIGH-PASS FILTER 600 Hz CUT-OFF

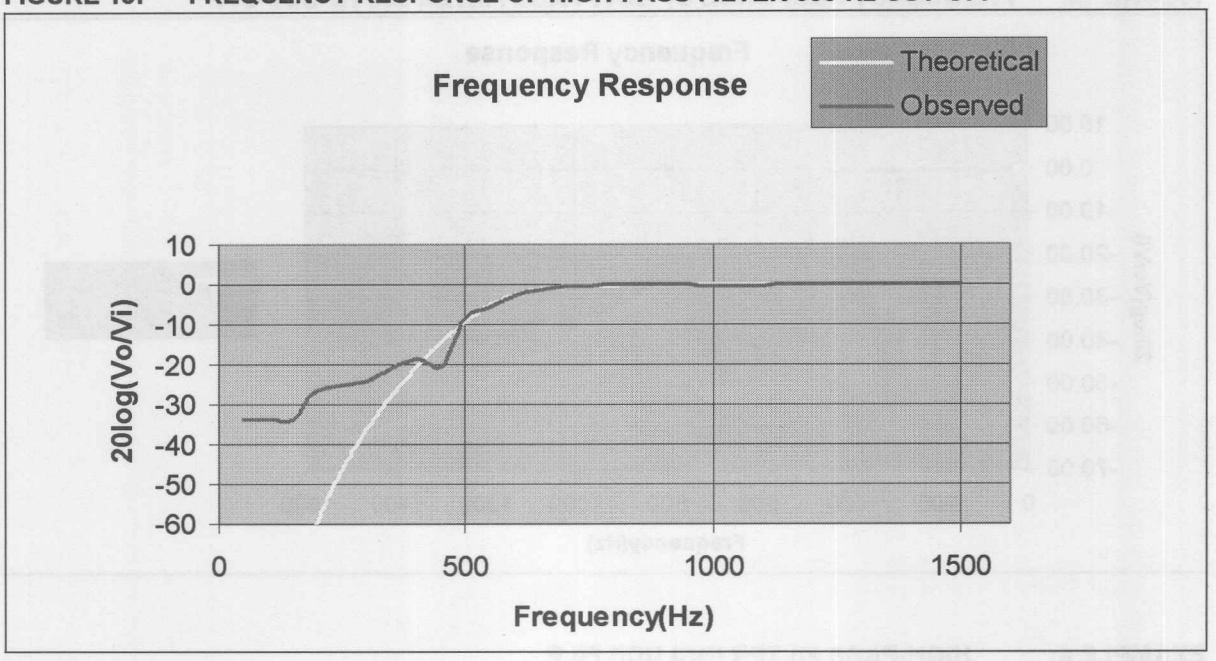


Table 9 shows the execution time and the corresponding maximum sampling frequency and MIPs for the above filters.

TABLE 9: IIR FILTER PERFORMANCE STATISTICS

Filter	Execution Time	Average Execution Time Per BIQUAD Section	Maximum Sampling Frequency Possible	MIPs at 8 kHz Sampling Rate
Low-pass 500 Hz cut-off	60.2 µs	20.07 µs	16.611 kHz	2.41
High-pass 600 Hz cut-off	61.2 µs	20.4 µs	16.339 kHz	2.448

Note: Processor clock frequency = 20 MHz.

The memory requirements for the above filters are as follows.

TABLE 10: IIR FILTER MEMORY REQUIREMENTS

Filter	Program Memory Locations	Data Memory Locations
Low-pass 500 Hz cut-off	301	34
High-pass 600 Hz cut-off	306	34

CONCLUSION

The software modules developed for FIR and IIR digital filters have been optimized for the execution speed of the PIC18 family of microcontrollers. For example, only one quarter of the available 10 MIPs can be used for filtering if a signal is sampled at the rate of 8 kHz, when a 6th order IIR filter or a 31 tap FIR filter is realized. The remaining MIPs are available to execute other applications as required by the user. Moreover, the software is linkable and relocatable.

Compile time options are provided to easily change the number of filter taps (in case of FIR), or the order of the filter (in case of IIR), sampling frequency, etc. Therefore, the software modules can easily be used for a variety of applications, such as filtering various kinds of sensor outputs (where the sampling rate may be much less than 8 kHz), as well as detecting some selected frequency components present in a speech signal.

APPENDIX A: DIGITAL FILTER BASICS

In signal processing, signals are often encountered that contain unwanted information, such as random noise or interference, or there is a need to selectively extract a signal of interest merged with several other signals. Filters are used in these situations to separate the signals of interest from others.

Filters can be analog or digital. Analog filters use electronic circuits made from components, such as resistors, capacitors, inductors and so forth, to produce the required filtering effect. At all stages, the signal being filtered is an electrical voltage or current, which is the direct analogue of the physical quantity (e.g., a sound or video signal or transducer output) involved.

A digital filter uses a digital processor to perform numerical calculations on sampled values of the signal. The processor may be a general purpose computing machine, such as a PIC18 microcontroller or a specialized DSP chip.

So that an analog signal in the form of voltage or current can be filtered, it must be converted to digital numbers to perform computations. Therefore, an Analog-to-Digital Converter (ADC) is used to transform the voltage or current to numbers. This process of converting the signal to digital numbers involves two processes, known as Sampling and Quantization.

Sampling is the process of sensing the analog values at discrete time intervals. Quantization is the process of converting the sensed analog voltage to discrete values. Note that with quantization, the signal values are approximated to a finite set of values. The value obtained after sampling and quantization is referred to as Sample Value. Do we need to convert all instantaneous values of an analog signal to numbers? If the answer is yes, this is an impossible task. Fortunately the answer is no, provided the signal satisfies certain conditions. The Nyquist Sampling theorem states this condition.

According to this theorem, if the signal has frequency components only up to a frequency of F Hz, then the signal must be sampled at 2F times/sec or more to prevent loss of signal information.

After sampling and quantization, the signal is in the form of a sequence of numbers.

Let us now examine the effect of computation on the sequence of numbers. Consider a sequence of numbers ..., 1, 2, 1, 2, 1, 2, 1, ... This sequence represents a triangular wave analog signal. If each sample value is multiplied by the value k, this results in the sequence ...., k, 2k, k, 2k, k, ... By performing this computation, the pattern of the sequence is not altered; however, the values are being scaled, which may result in amplification (for k>1) or attenuation (k<1).

The above computation is an example of an All-pass filter. This filter will pass all frequencies; therefore, the input pattern is repeated as it is.

The input to output sample relation in the above example can be represented by Equation A-1.

EQUATION A-1: TAP LENGTH 1 FILTER

$$y[n] = k * x[n]$$

Where $x[n]$ represents nth input sample and $y[n]$ represents nth output sample.

In this equation, we are considering only one input sample to compute the present output sample. This is a filter of *tap length 1*. There is only one filter coefficient whose value is k.

In Equation A-1 only one sample value was used. Consider the following example, where two previous samples will be used with one present sample, as shown in Equation A-2. Because three input values are being taken for computation, this is a filter of *tap length 3*.

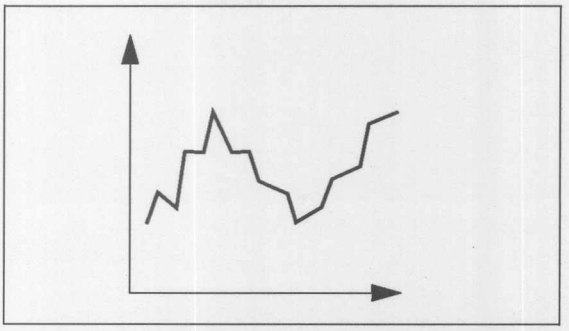
EQUATION A-2: TAP LENGTH 3 FILTER

$$y[n] = \frac{1}{3}x[n-2] + \frac{1}{3}x[n-1] + \frac{1}{3}x[n]$$

The result of Equation A-2 gives a sequence of numbers that will fluctuate with the average value of the previous two samples and one present sample. If there is riding noise over a signal envelope, as shown in Figure A-1, then this computation can remove the riding noise by means of averaging, as shown in Figure A-2.

The filter coefficients for this example are 1/3, 1/3, 1/3.

FIGURE A-1: SIGNAL ENVELOPE WITH NOISE



Instead of two previous samples, if a large number of previous samples are considered, then the output sequence of numbers will almost remain constant, which represents the D.C. component of the input signal. This computation results in a low-pass filter.

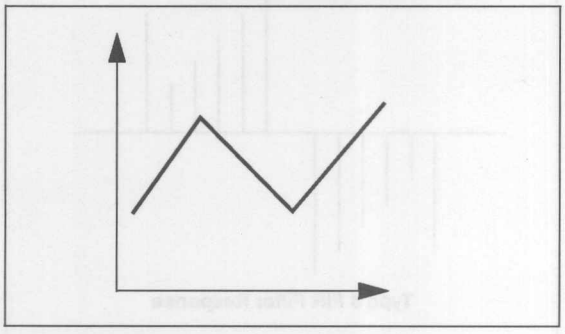
At this point, two specific digital filter examples have been considered. Now we will generalize the above examples of digital filters.

A digital filter, in its most general form, takes in an input sequence of numbers x[n], performs computations on these numbers and outputs results of these computations as another sequence of numbers y[n]. In general, the output sequence of numbers can be expressed as shown in Equation A-3.

EQUATION A-3: FILTER OUTPUT SEQUENCE

$$y[n] = a_0x[n] + a_1x[n-1] + ... + a_Mx[n-M] + b_1y[n-1] + b_2y[n-2] + ... + b_Ny[n-N]$$

FIGURE A-2: SIGNAL ENVELOPE WITH NOISE AVERAGED OUT



Note that along with the present and previous input samples, we have included the previous output samples (i.e., y[n-1], y[n-2] ...) also in the computation of present output sample. This is further discussed in "Types of Digital Filters".

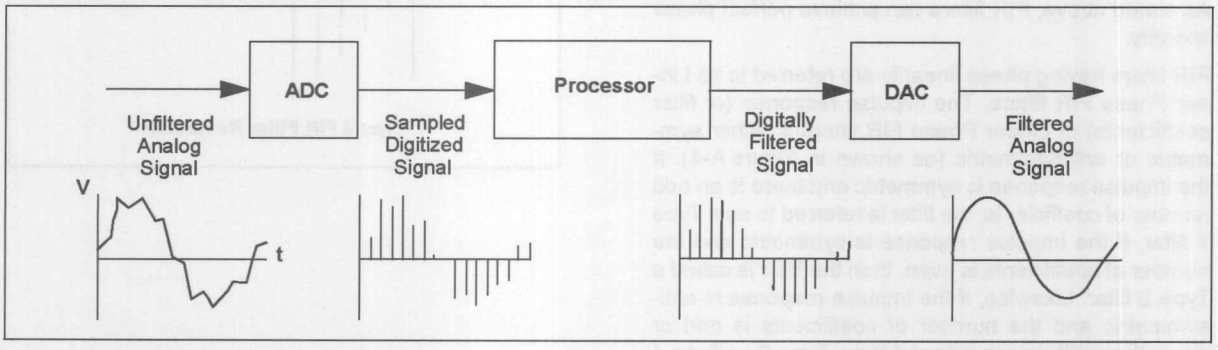
To form an equivalent to an analog filter, the input sequence of numbers is derived by passing the analog signal (to be filtered) through an ADC, as discussed earlier. Normally, an anti-aliasing filter (must be an analog filter) precedes the ADC to remove all frequencies above the 1/2 of sampling frequency. The output sequence y[n] is converted to an analog signal by a DAC, followed by a low-pass filter. A general form of signal processing using digital filter is illustrated in Figure A-3. The computation performed on the sampled values decides the characteristics of the filter.

Digital filters offer the following advantages over their analog counterparts:

1. A digital filter is programmable (i.e., its operation is determined by program stored in a processors memory). This means the digital filter can easily be changed without affecting the circuitry (hardware). An analog filter can only be changed by redesigning the filter circuit.
2. Digital filter performance is repeatable and reliable.
3. Requires no tuning components.
4. Free from component drift.
5. Does not require precision components.
6. Superior performance.
7. Digital filters are very versatile in their ability to process signals in a variety of ways; this includes the ability of some types of digital filters to adapt to changes in the characteristics of the signal.

However, the dynamic range of digital filters is drastically low compared to analog filters because of the finite quantization of levels.

FIGURE A-3: SIGNAL PROCESSING USING A DIGITAL FILTER



Types of Digital Filters

Generally, y[n] is computed as the sum of weighed present and previous input samples and previous output samples, as shown in Equation A-4.

EQUATION A-4: FILTER COMPUTATION

$$y[n] = a_0x[n] + a_1x[n-1] + ... + a_Mx[n-M] + b_1y[n-1] + b_2y[n-2] + ... + b_Ny[n-N]$$

Where $a_0, a_1, ... a_M$ and $b_1, b_2, ... b_N$ are constants and referred to as filter coefficients. M+1 and N are the number of input and output samples used for computation.

If b_1 through b_N are all zeros, then y[n] does not depend on the previous output samples (i.e., there is no feedback). In this case, this type of filter is termed as a Finite Impulse Response (FIR) filter. Since there is no feedback term if the input sequence stops (i.e., x[n]'s become zeros), then y[n]'s also will become zeros after some delay. If any one of the coefficients b_1 through b_N are non-zero, the filter is called an Infinite Impulse Response (IIR) filter. For the FIR filter, the sequence of coefficients $a_0, a_1, .. a_M$ also represent the response of the filter for a unit impulse (also called an impulse response).

The advantages of FIR filters are:

- They can be designed to have linear phase response with respect to frequency, whereas IIR filters do not have linear phase response.
- They are always stable, unlike IIR filters.

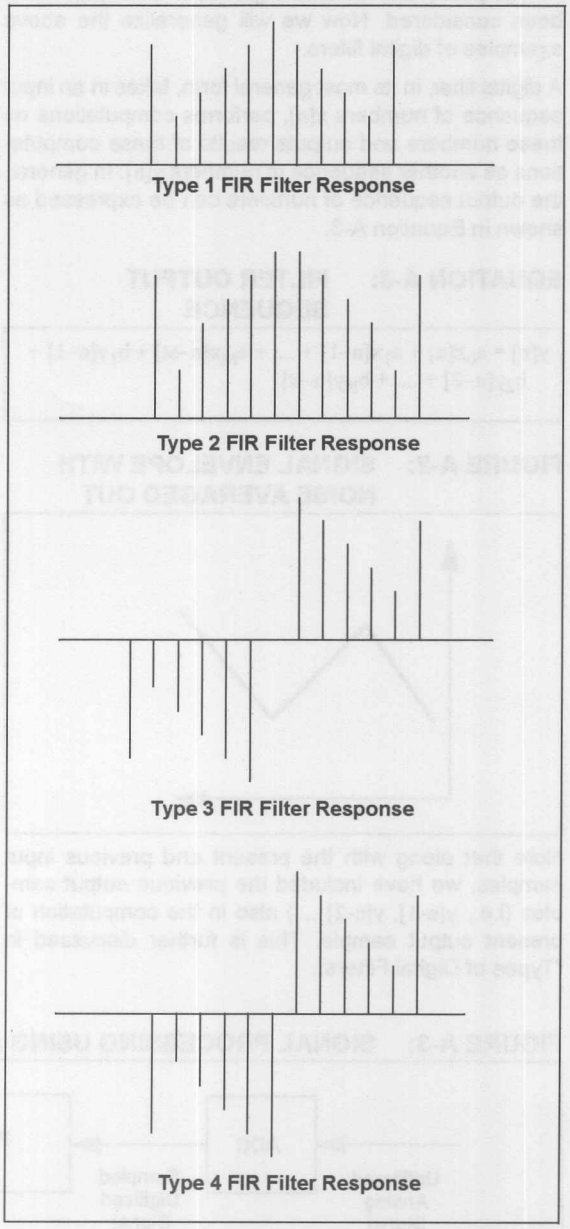
The disadvantages of FIR filters over IIR filters are:

- FIR filters take relatively more memory and computation time.
- FIR filters cannot give sharper cut-off than IIR filters for the same number of filter coefficients.

As stated above, FIR filters can achieve perfect phase linearity.

FIR filters having phase linearity are referred to as Linear Phase FIR filters. The impulse response (or filter coefficients) of Linear Phase FIR filters is either symmetric or anti-symmetric (as shown in Figure A-4). If the impulse response is symmetric and there is an odd number of coefficients, the filter is referred to as a Type 1 filter. If the impulse response is symmetric and the number of coefficients is even, then the filter is called a Type 2 filter. Likewise, if the impulse response is anti-symmetric and the number of coefficients is odd or even, these filters are referred to as Type 3 or Type 4 filters, respectively.

FIGURE A-4: FILTER RESPONSE TYPES



FIR Filter Design Methods

The following three methods are commonly used for FIR filter design:

- Fourier Series
- Frequency Sampling
- Remez Exchange

FOURIER SERIES

This method is based on the fact that the frequency response of a digital filter is periodic. Therefore, a digital filter can be expanded in the form of Fourier Series. Because this series contains an infinite number of terms, the expansion is truncated to a finite number of terms. The coefficients of these terms are then used as filter coefficients. However, because of truncation, the filter characteristics may change. To prevent this, the filter response must be determined using these coefficients and compared with the expected response. If the results are not satisfactory, the number of terms are increased and the iteration is repeated until the expected response is achieved.

Because of the finite number of terms of expansion, the frequency response exhibits overshoots and undershoots near the regions of cut-off frequency, which is known as Gibbs phenomenon. To avoid this, the truncated coefficients are multiplied by a set of coefficients known as Window Function (e.g., Kaiser, Bartlett, Hamming, and so on).

FREQUENCY SAMPLING

In the Fourier Series method, the desired frequency response is specified in continuous frequency domain. In contrast, in the Frequency Sampling method, the desired frequency response is specified in discrete frequency domain. Inverse discrete Fourier transform is then used to obtain the filters impulse response or filter coefficients.

REMEZ EXCHANGE

This method minimizes the maximum error between the desired frequency response and the actual frequency response. Filters designed with this method meet the given specification with the lowest filter order.

IIR Filter Design Methods

The common method of designing an IIR filter is as follows. The transfer function H(S) of an analog filter is derived for the required specifications. This transfer function is then converted to Z domain (H(Z)), which represents the Z transform of the transfer function of the desired digital filter. The conversion from S domain to Z domain can be done by any of the following methods:

- Impulse Invariant
- Step Invariant
- Bilinear Transformation
- Matched Z

IMPULSE INVARIANT

In this method, the S domain transfer function is converted to time domain impulse response *f(t)*. From *f(t)*, the digital filter impulse response is derived, where the value of the digital filter impulse response is equal to the value of the analog filter impulse response at time intervals T, i.e., $h[n]=f(nT)$.

The impulse response is then converted to transfer function in Z domain by taking Z transform of $h(n)$, as shown in Equation A-5.

EQUATION A-5: IMPULSE RESPONSE CONVERSION

$$H(Z) = \sum_{0}^{\infty} h[k]Z^{-k}$$

STEP INVARIANT

This method is similar to the Impulse Invariant method, except the step response is used instead of impulse response.

BILINEAR TRANSFORMATION

The S in the continuous transfer function is substituted with the Z expression below to create the transfer function of the digital filter, $S = 2(Z-1) / T(Z+1)$.

The resulting expression for H(Z) is independent of T because T gets cancelled out.

MATCHED Z

In this method, the poles and zeros of the transfer function H(S) are mapped directly to poles and zeros of the transfer function H(Z) by substituting terms (S+a) with $1-e^{-aT}Z^{-1}$. For example, a pole/zero at s = -a is mapped to pole/zero at e^{-aT}.

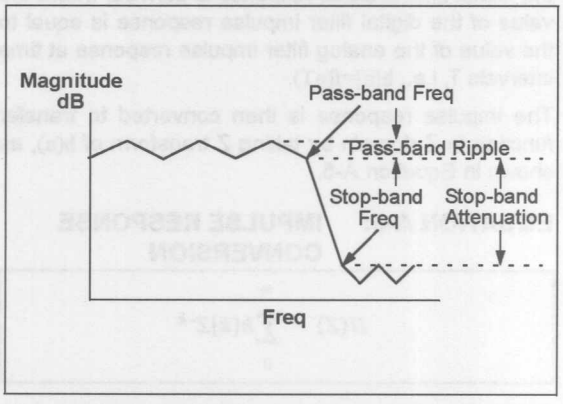
APPENDIX B: FIR FILTER INCLUDE FILE PREPARATION EXAMPLE

In the following example, a freeware program (see Appendix F) was used to design coefficients for a FIR low-pass filter with the following specifications:

- Sampling frequency: 8000 Hz
- Pass-band frequency: 3000 Hz
- Stop-band frequency: 3300 Hz
- Pass-band ripple: 2 db
- Stop-band attenuation: 40 db

Figure B-1 provides a visual representation of the relationships between each of the parameters and the filter that is being designed.

FIGURE B-1: FILTER PARAMETER RELATIONSHIPS



The required specifications were entered in the coefficient generation freeware. Table B-1 lists the resulting coefficients.

TABLE B-1: INITIAL FILTER COEFFICIENTS

H[0]	0.004667
H[1]	0.056084
H[2]	0.018875
H[3]	-0.026130
H[4]	0.025119
H[5]	-0.016081
H[6]	-0.001788
H[7]	0.023866
H[8]	-0.040989
H[9]	0.042084
H[10]	-0.019634
H[11]	-0.027082
H[12]	0.089986
H[13]	-0.153881
H[14]	0.201432
H[15]	0.781025
H[16]	0.201432
H[17]	-0.153881
H[18]	0.089986
H[19]	-0.027082
H[20]	-0.019634
H[21]	0.042084
H[22]	-0.040989
H[23]	0.023866
H[24]	-0.001788
H[25]	-0.016081
H[26]	0.025119
H[27]	-0.026130
H[28]	0.018875
H[29]	0.056084
H[30]	0.004667

The initial filter coefficients received are then translated, so that the maximum value maps to 127 and the translated values are rounded to the nearest integer. Table B-2 lists the translated filter coefficients and the corresponding HEX code. The HEX values are then entered into the include file and used to implement the FIR filter.

TABLE B-2: TRANSLATED FILTER COEFFICIENTS

Filter Coefficient	Translated Integer Value	HEX Code
H[0]	1	01
H[1]	9	09
H[2]	3	03
H[3]	-4	FC
H[4]	4	04
H[5]	-3	FD
H[6]	0	00
H[7]	4	04
H[8]	-7	F9
H[9]	7	07
H[10]	-3	FD
H[11]	-4	FC
H[12]	15	0F
H[13]	-25	E7
H[14]	33	21
H[15]	127	7F
H[16]	33	21
H[17]	-25	E7
H[18]	15	0F
H[19]	-4	FC
H[20]	-3	FD
H[21]	7	07
H[22]	-7	F9
H[23]	4	04
H[24]	0	00
H[25]	-3	FD
H[26]	4	04
H[27]	-4	FC
H[28]	3	03
H[29]	9	09
H[30]	1	01

The following include file was created using the HEX values from Table B-2.

EXAMPLE B-1: FIR FILTER INCLUDE FILE

```
CONSTANT num_of_taps=D'31'    ;Enter the filter tap length here
CONSTANT dist_to_last_tap=num_of_taps-1
CONSTANT dist_to_prv_to_last_tap=num_of_taps-2
;define filter coeffs here
CONSTANT coeff0=0x01          ;corresponds to the latest sample
CONSTANT coeff1=0x09
CONSTANT coeff2=0x03
CONSTANT coeff3=0xfc
CONSTANT coeff4=0x04
CONSTANT coeff5=0xfd
CONSTANT coeff6=0x00
CONSTANT coeff7=0x04
CONSTANT coeff8=0xf9
CONSTANT coeff9=0x07
CONSTANT coeff10=0xfd
CONSTANT coeff11=0xfc
CONSTANT coeff12=0x0f
CONSTANT coeff13=0xe7
CONSTANT coeff14=0x21
CONSTANT coeff15=0x7f
CONSTANT coeff16=0x21
CONSTANT coeff17=0xe7
CONSTANT coeff18=0x0f
CONSTANT coeff19=0xfc
CONSTANT coeff20=0xfd
CONSTANT coeff21=0x07
CONSTANT coeff22=0xf9
CONSTANT coeff23=0x04
CONSTANT coeff24=0x00
CONSTANT coeff25=0xfd
CONSTANT coeff26=0x04
CONSTANT coeff27=0xfc
CONSTANT coeff28=0x03
CONSTANT coeff29=0x09
CONSTANT coeff30=0x01     ;corresponds to the oldest sample
```

APPENDIX C: IIR FILTER INCLUDE FILE PREPARATION EXAMPLE

For this example, we'll assume that we want an IIR Butterworth high-pass filter with the following specifications:

- Sampling frequency: 8000 Hz
- Cut-off frequency: 500 Hz
- Stop-band frequency: 200 Hz
- Pass-band ripple: 1 db
- Stop-band attenuation: 40 db

The coefficients in this example were designed using public domain freeware (see Appendix F).

Using the above specifications, the following coefficients are obtained for a sixth order filter.

TABLE C-1: FIR FILTER COEFFICIENTS

Section	Coefficient	Decimal Value
1	a1_0	0.5000000000000
	a1_1	-1.000000000000
	a1_2	0.5000000000000
	b1_1	-1.408666209103
	b1_2	0.5005623325360
2	a2_0	0.5000000000000
	a2_1	-1.000000000000
	a2_2	0.5000000000000
	b2_1	-1.509681989763
	b2_2	0.6081680055271
3	a3_0	0.5000000000000
	a3_1	-1.000000000000
	a3_2	0.5000000000000
	b3_1	-1.723786170135
	b3_2	0.8362395431481

Determining Peak Gain

Using the coefficients for each `BIQUAD` section, we can now determine the peak gains `gc1`, `gc2`, and `gc3` by plotting frequency response. The frequency response plots shown in Figure C-1, Figure C-2 and Figure C-3 were created using a freeware program (see Appendix F).

As seen from these figures, the peak gains for sections 1, 2, and 3 are `gc1=0.6941`, `gc2=0.6475` and `gc3=1.139`, respectively.

FIGURE C-1: SECTION 1 FREQUENCY RESPONSE

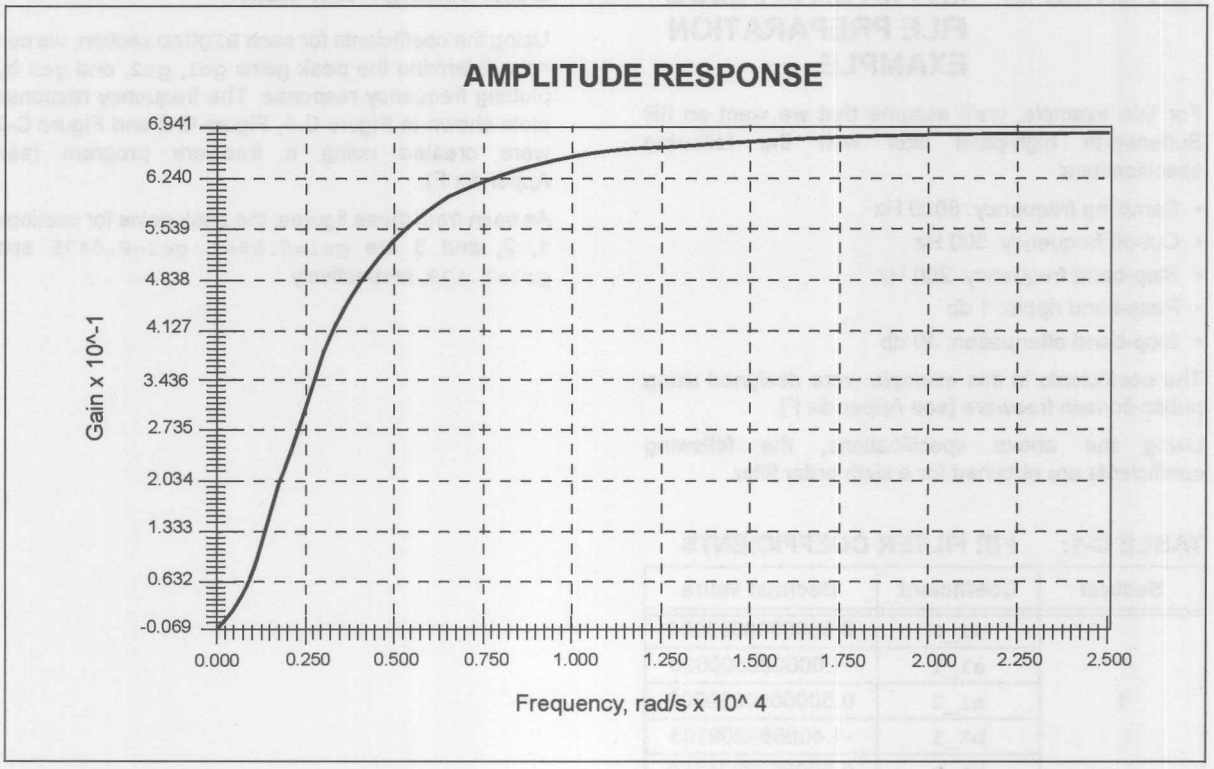


FIGURE C-2: SECTION 2 FREQUENCY RESPONSE

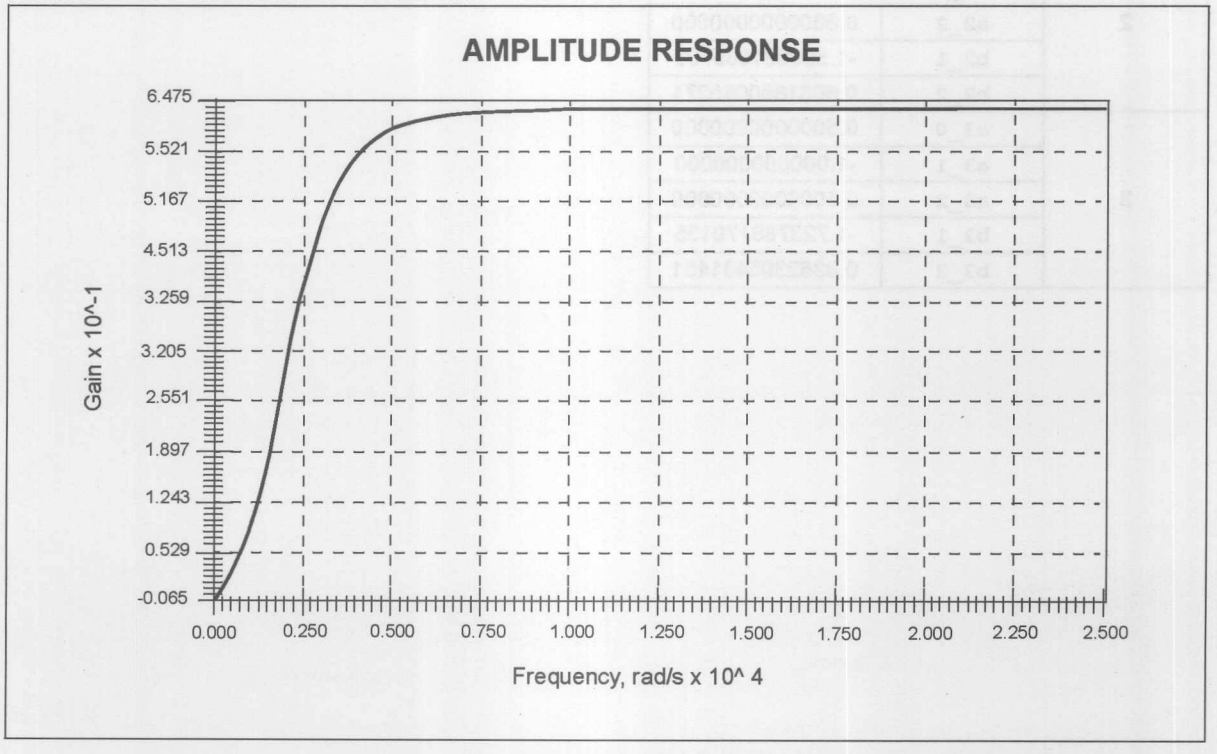
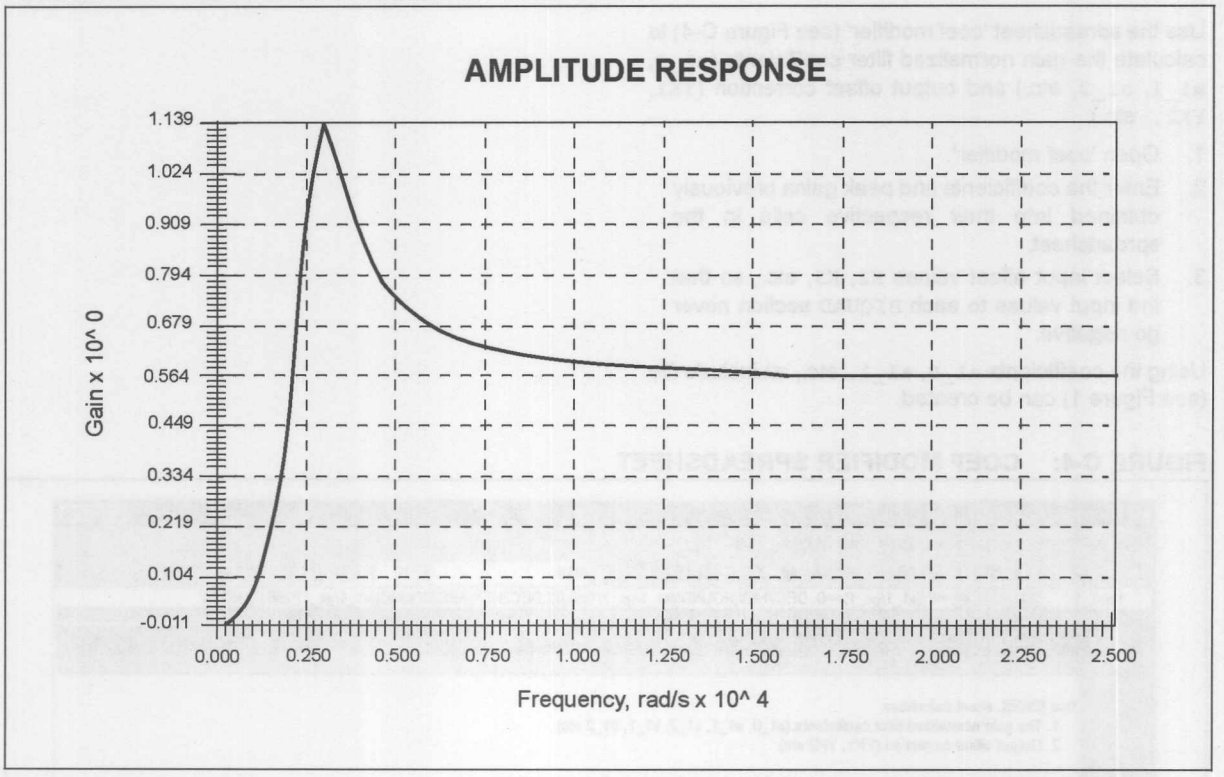


FIGURE C-3: SECTION 3 FREQUENCY RESPONSE



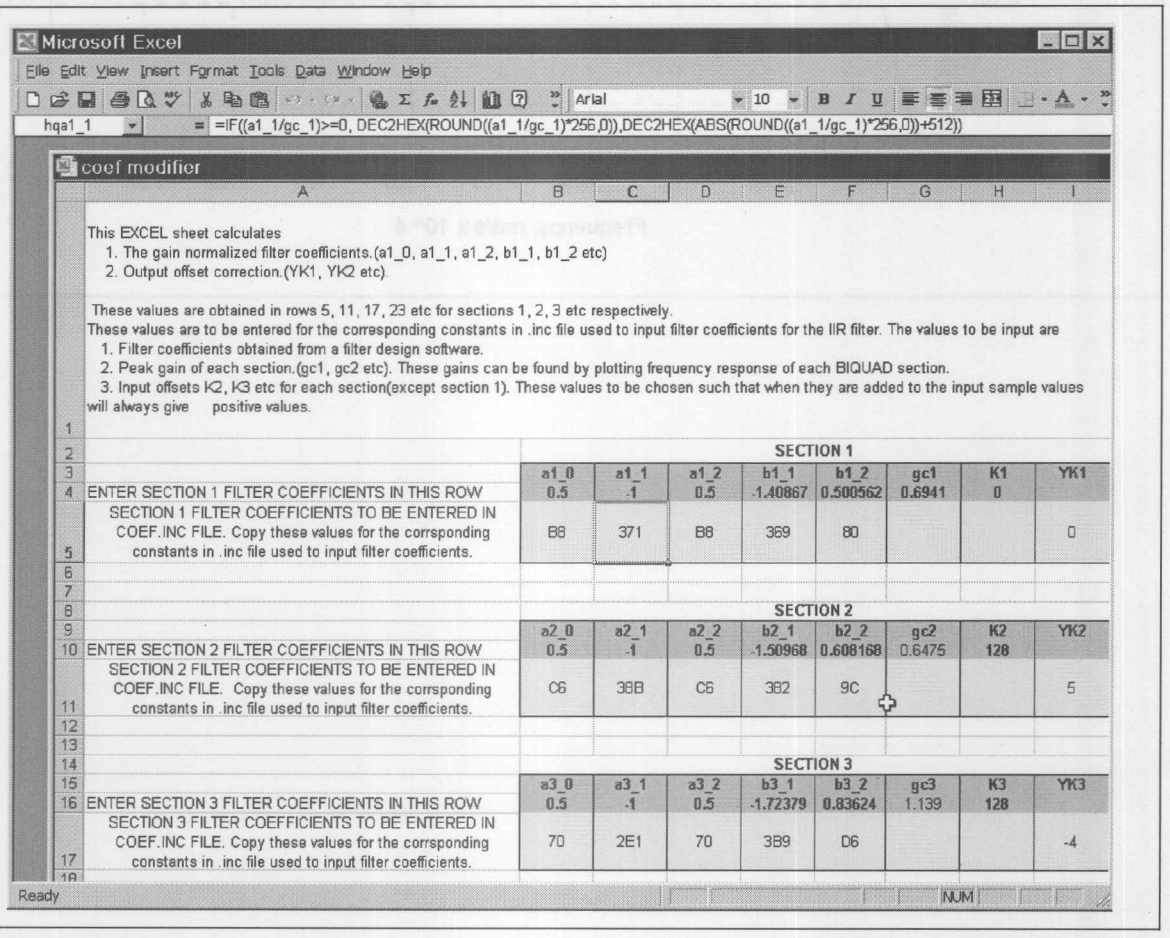
Coef Modifier

Use the spreadsheet 'coef modifier' (see Figure C-4) to calculate the gain normalized filter coefficients (a1_0, a1_1, a1_2, etc.) and output offset correction (YK1, YK2, etc.).

1. Open 'coef modifier'.
2. Enter the coefficients and peak gains previously obtained into their respective cells in the spreadsheet.
3. Select input offset values K2, K3, etc., so that the input values to each BIQUAD section never go negative.

Using the coefficients a1_0, a1_1, etc., an include file (see Figure 1) can be created.

FIGURE C-4: COEF MODIFIER SPREADSHEET



EXAMPLE C-1: HIGH-PASS BUTTERWORTH FILTER INCLUDE FILE

```
;*****************************************************************
;High pass Butterworth filter
;sampling frequency 8000 Hz
;Cut-off frequency 500 Hz
;Stop band frequency 200 Hz
;Pass band ripple 1 dB
;Stop band attenuation 40 dB
;*****************************************************************
;specify the number of biquad sections in the following line
CONSTANT NUMBER_OF_SECTIONS=3
;******************SECTION 1*************************************
CONSTANT a1_0=0XB8
CONSTANT a1_1=0X371
CONSTANT a1_2=0XB8

CONSTANT b1_1=0x369
CONSTANT b1_2=0X80
CONSTANT YK1=D'0' ;enter this value in decimal representation only
;******************SECTION 2*************************************
CONSTANT a2_0=0XC6
CONSTANT a2_1=0X38B
CONSTANT a2_2=0XC6

CONSTANT b2_1=0X382
CONSTANT b2_2=0X9C
CONSTANT K2=128
CONSTANT YK2=D'5' ;enter this value in decimal representation only
;******************SECTION 3*************************************
CONSTANT a3_0=0X70
CONSTANT a3_1=0X2E1
CONSTANT a3_2=0X70

CONSTANT b3_1=0X3B9
CONSTANT b3_2=0XD6
CONSTANT K3=128
CONSTANT YK3=D'-4' ;enter this value in decimal representation only
```

APPENDIX D: EXAMPLE PROGRAMS

FIR Filter Program (`expl_fir.asm`)

```
;FIR filter program example expl_fir.asm
       list p=18c452 ;specifies the processor used.
       #include<p18c452.inc>
       #include<coef.inc>            ;This defines filter characteristics.
                                      ;Enter number of taps and filter coefficients in
                                      ;this file.
CONSTANT IN_PORT=RA1                  ;Enter the input port used. Options available
                                      ;are RA1,RA2,RA3,RA4,RA5,RA6
CONSTANT OUT_PORT_HIGH=PORTD          ;Enter the output port used for Most
                                      ;significant byte. Options available are
                                      ;PORTB, PORTC and PORTD.
CONSTANT OUT_PORT_LOW=PORTB           ;Enter the output port used for least
                                      ;significant byte. Options available are
                                      ;PORTB, PORTC and PORTD.
                                      ;Do not use the same port used for
                                      ;OUT_PORT_HIGH.
CONSTANT INPUT=ADRESH                 ;Enter the source register of I/P samples to
                                      ;the filter
;*****************************************************************************
;The value assigned to this constant 'num_of_mulacc' determines the number of MULACC
;routines used in software loop. The idea of providing this is to give the user a
;flexibility for a trade-off between number of program memory locations used and the
;execution time of interrupt service routine. Higher the value of this number lower
;the execution time and hence higher the maximum usable sampling frequency. The value
;of this constant can range from 1 to num_of_taps.
CONSTANT num_of_mulacc=D'31'
;*****************************************************************************
CONSTANT sample_freq=D'8000'          ;Enter the desired sample frequency
CONSTANT clock_freq=D'20000000'       ;Enter the processor clock frequency
       #include port.inc              ;sets up ports
                                      ;Note:- Pins RA0-RA6 defined as analog i/p
                                      ;ports in this file. User can modify this file
                                      ;if any of RA's is to be used as digital i/o's
       #include fir_buf.inc           ;Defines buffer spaces for FIR filter
;-----------------------------------------------------------------------------
;user memory assignments
;-----------------------------------------------------------------------------
       #include fir_mac.inc           ;includes FIR filter macros
       #include peri.inc              ;peripheral initialization macros
       #include int.inc               ;interrupt settings macro

       CODE     0x0
rst
       goto     start

int_hi CODE     0x8
       goto     int_service_hi
```

```
int_low CODE    0x18
        goto    int_service_low

main    CODE

start
        INIT_PERIPHERALS            ;Initializes peripherals
;*************************************************************************
;user can enter code here to set interrupts he is using
;Note: interrupt priority is enabled and all user interrupts should be assigned low
;priority.
;*************************************************************************

        SET_INTR_FILTER             ;Sets interrupt settings for filter
        INIT_FILTER                 ;Initializes filter buffers.

        bsf     T1CON,TMR1ON        ;Now that every thing is set timer is switched
                                    ;on now to begin sampling.
;*************************************************************************
;user's code can be entered here
;*************************************************************************

        goto $

int_service_hi
        bcf     PIR1,ADIF           ;clears A/D interrupt flag

        FIR_FILTER                  ;Filters the signal supplied through INPUT.
                                    ;filtered o/p (24 bit 2's complement)
                                    ;available in locations output_most, output_middle
                                    ;and output_least.

        rlcf    output_least,W      ;filtered o/p shifted left to get a gain of 2
        movwf   OUT_PORT_LOW        ;and output on output ports
        rlcf    output_middle,W     ;
        movwf   OUT_PORT_HIGH       ;
        retfie  FAST                ;
;*************************************************************************
int_service_low
;Users  interrupt service routine
        retfie                      ;
;*************************************************************************

        END
```

IIR Filter Program (expl_iir.asm)

```
;IIR filter program example expl_iir.asm
       list    p=18c452              ;specifies the processor used.
       #include<p18c452.inc>         ;
       #include<coef.inc>            ;This defines filter characteristics. Enter number
                                     ;of BIQUAD sections used, filter coefficients for
                                     ;each BIQUAD section, input offsets(K's) and output
                                     ;corrections (YK's) in this file.
CONSTANT IN_PORT=RA1                 ;Enter the input port used. Options available are
                                     ;RA1,RA2,RA3,RA4,RA5,RA6
CONSTANT OUT_PORT_HIGH=PORTD         ;Enter the output port used for Most significant
                                     ;byte. Options available are PORTB, PORTC and
                                     ;PORTD.
CONSTANT OUT_PORT_LOW=PORTB          ;Enter the output port used for least significant
                                     ;byte. Options available are PORTB, PORTC and
                                     ;PORTD.
                                     ;Do not use the same port used for OUT_PORT_HIGH.
CONSTANT INPUT=ADRESH                ;Enter the source register of I/P samples to the
                                     ;filter
CONSTANT sample_freq=D'8000'         ;Enter the desired sample frequency
CONSTANT clock_freq=D'20000000'      ;Enter the processor clock frequency

       #include  port.inc           ;sets up ports
                                     ;Note:- Pins RA0-RA6 defined as analog i/p ports in
                                     ;this file. User can modify this file if any of
                                     ;RA's is to be used as digital i/o's
       #include iir_buf.inc          ;defines buffers for filter
;-------------------------------------------------------------------------------
;user memory assignments
;-------------------------------------------------------------------------------
       #include iir_mac.inc          ;includes iir filter macros
       #include peri.inc            ;peripheral initialization macros
       #include int.inc             ;interrupt settings

       CODE     0x0
rst
       goto     start

int_hi CODE     0x8
       goto     int_service_hi

int_low CODE    0x18
       goto     int_service_low

main   CODE

start
       INIT_PERIPHERALS             ;Initializes peripherals
;*******************************************************************************
;user can enter code here to set interrupts he is using note interrupt priority is
;enabled and all user interrupts should be assigned low priority.
;*******************************************************************************

       SET_INTR_FILTER              ;Sets interrupt settings for filter
       INIT_FILTER                  ;Initializes filter buffers.

       bsf     T1CON,TMR1ON         ;Now that every thing is set timer is switched on
                                    ;now to begin sampling.
;*******************************************************************************
;user's code can be entered here
;*******************************************************************************

       goto    $

int_service_hi
```

```
here    bcf      PIR1,ADIF          ;clears A/D interrupt flag

        IIR_FILTER                  ;Filters the signal supplied through INPUT.
                                    ;filtered o/p (32 bit 2's complement) available in
                                    ;locations sum, sum+1, sum+2 and sum+3.
there   rlcf     sum+2,W            ;filtered o/p shifted left to get a gain of
        movwf    OUT_PORT_LOW       ;2 and output on output ports
        rlcf     sum+1,W            ;
        movwf    OUT_PORT_HIGH      ;
        retfie   FAST               ;

;*****************************************************************************
int_service_low
;Users  interrupt service routine
        retfie                      ;
;*****************************************************************************

        END
```

APPENDIX E: FLOW CHARTS

FIGURE E-1: MAIN ROUTINE OF FIR FILTER

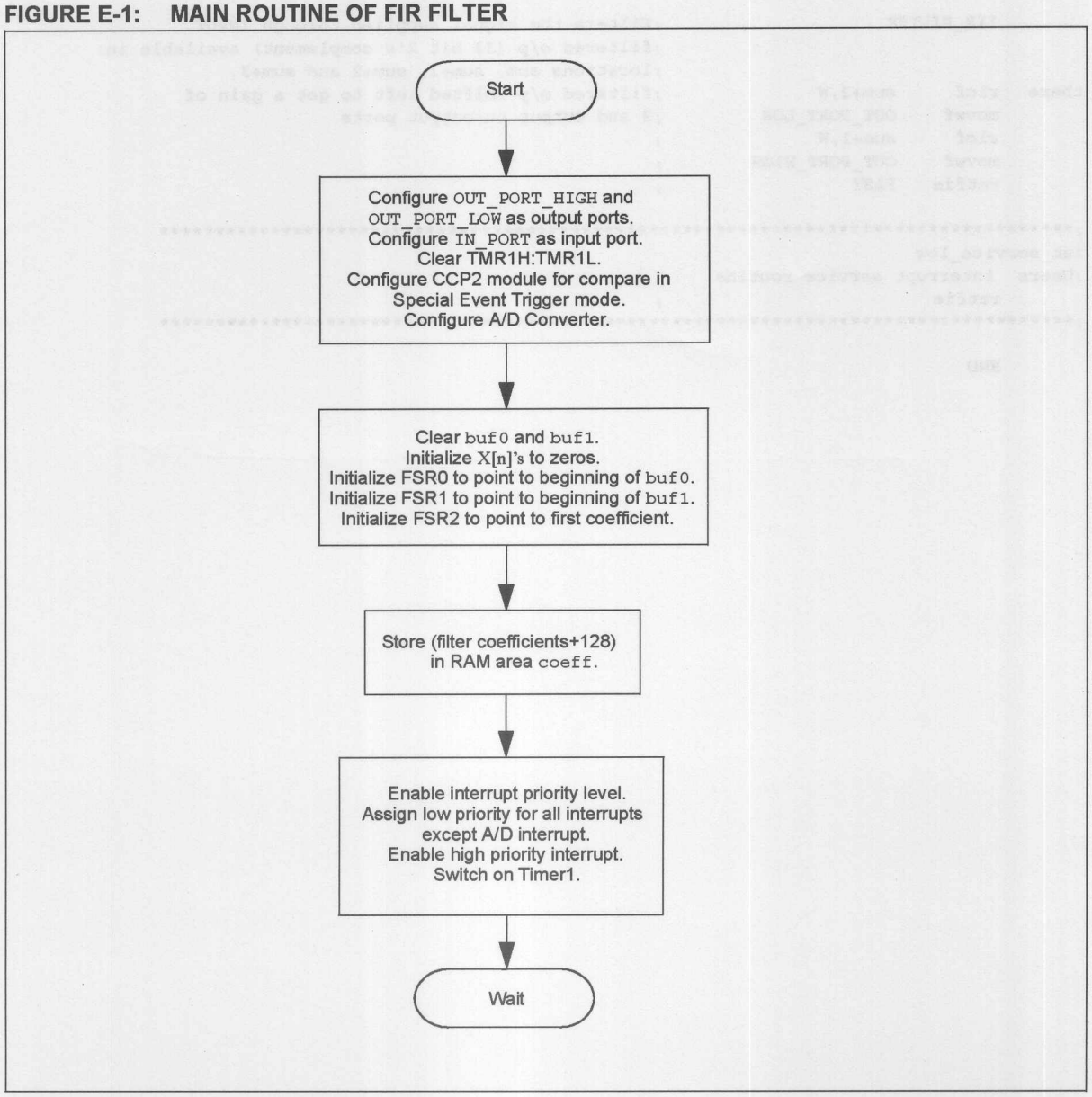


FIGURE E-2: INTERRUPT SERVICE ROUTINE OF FIR FILTER

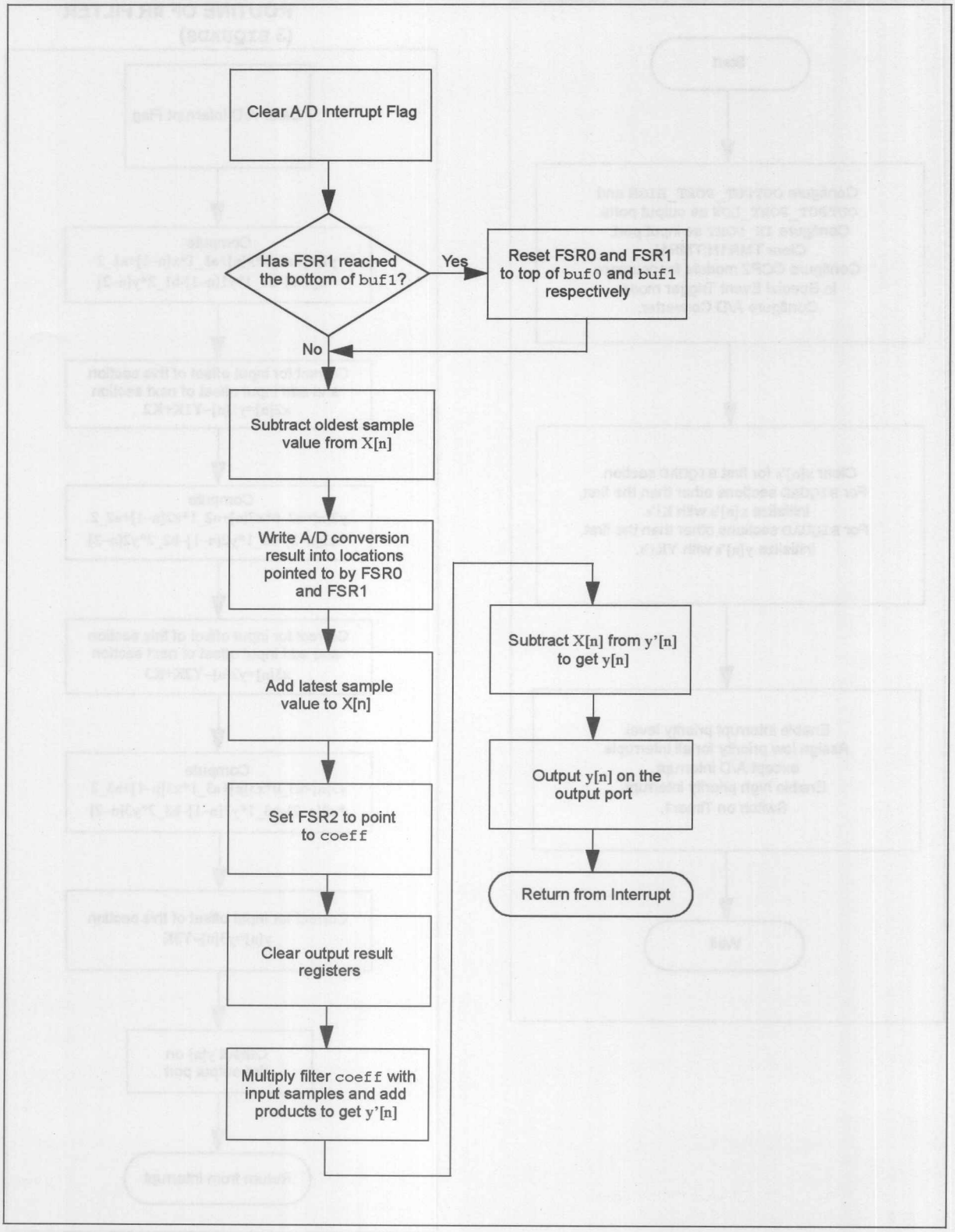


FIGURE E-3: MAIN ROUTINE OF IIR FILTER

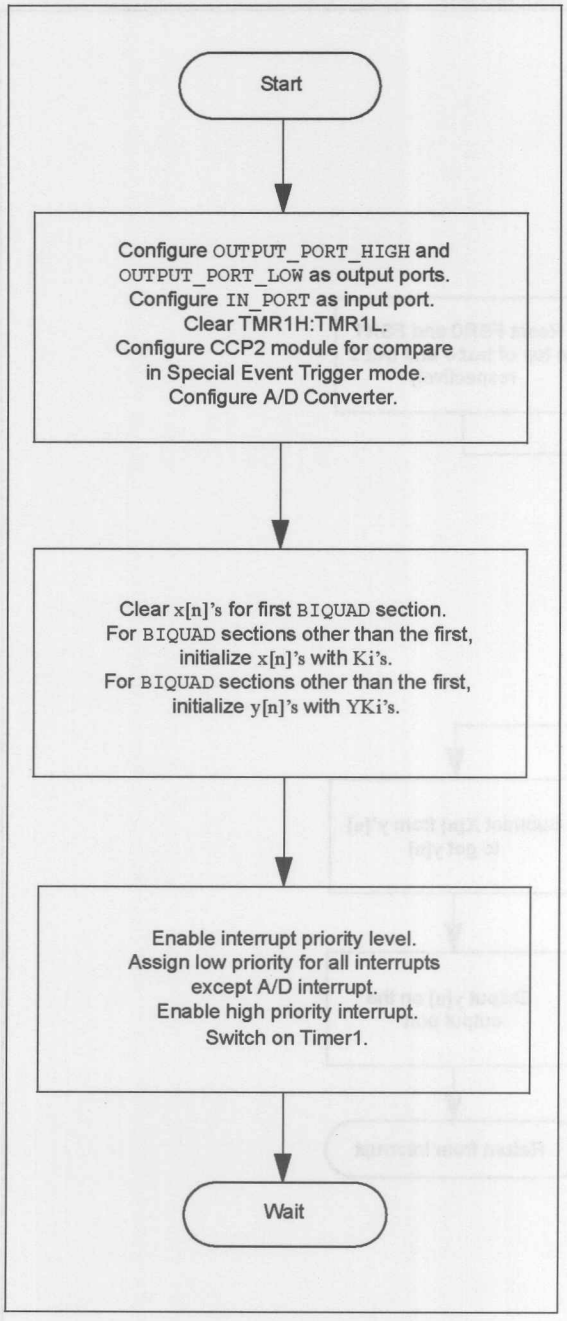
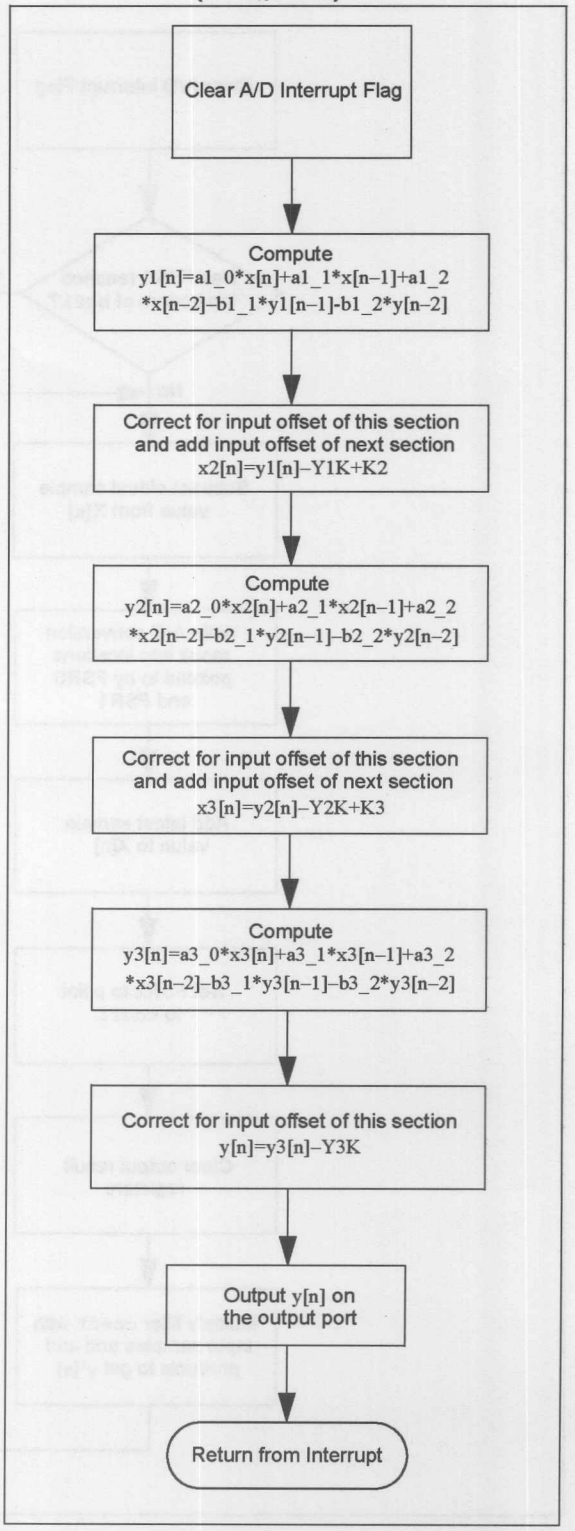


FIGURE E-4: INTERRUPT SERVICE ROUTINE OF IIR FILTER (3 BIQUADS)



APPENDIX F: REFERENCES

Following are the Worldwide web sites where digital filter design and coefficient generation freeware referenced in this application can be obtained (users assume any risk associated with using freeware from these sites):

- http://www.eliteeng.com/downloads.htm
  (Digital filter coefficient generation program)
- http://www.cmsa.wmin.ac.uk/filter_design.html
  (CMSA filter designer)
- http://www.hta-bi.bfh.ch/CPG/software/dsplay.html
  (Digital signal processing experimentation freeware)
- http://membres.lycos.fr/yannstrc/download.html
  (`BIQUAD` coefficient generator)
- http://www.ece.uvic.ca/~cathy/PCsoftware/
  (Digital filter designer/analyzer)

APPENDIX G: SOURCE CODE

Due to size considerations, the complete source code for this application note is not included in the text.

You can download the source code, which includes the spreadsheet 'coef modifier' from the Microchip web site at:

www.microchip.com

Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable".
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

DNV Certification, Inc. USA — ANSI-RAB QMS ACCREDITED — DNV MSC The Netherlands Accredited by the RvA — DNV — ISO 9001 / QS-9000 REGISTERED FIRM

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999 and Mountain View, California in March 2002. The Company's quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, non-volatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001 certified.